UQAC
UNIVERSITÉ DU QUÉBEC
À CHICOUTIMI

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MAXIME SOUCY-BOIVIN

B.Sc.A.

VÉRIFICATION DE PROPRIÉTÉS LTL DANS UN ENVIRONNEMENT HADOOP MAPREDUCE

AVRIL 2015

RÉSUMÉ

Dans le présent mémoire nous aborderons la vérification de propriétés LTL dans un environnement Hadoop MapReduce. Nous allons en premier parler de l'utilité d'une telle idée. Donc, nous allons définir notre source de données. Ensuite, nous définirons pourquoi ce type d'analyse est intéressant et ensuite les outils existants.

Par la suite, nous allons nous pencher sur la logique utilisée pour permettre d'écrire les faits ou informations à observer sur les données d'analyse. La logique utilisée est la logique temporelle linéaire et nous prendrons le temps de définir ce que sont les opérateurs contenus dans cette dernière. Sans oublier, nous parlerons de la structure des données utilisées et les avantages de cette dernière.

Nous étudierons le paradigme MapReduce qui permet d'encadrer le traitement selon une structure prédéfinie. Donc, nous allons présenter l'ensemble des phases avec une description permettant de bien identifier leur utilité. Ensuite, nous introduirons les outils permettant d'utiliser le paradigme MapReduce. C'est-à-dire de ce que sont Mr Sim et l'environnement Hadoop MapReduce. Nous allons décrire le fonctionnement distribué de ces environnements à travers les différentes phases. De plus, pour permettre de mieux comprendre ces environnements, nous présenterons des explications en profondeur sur des exemples concrets. Pour terminer, nous effectuerons une comparaison entre les deux outils.

Ensuite, nous parlerons de l'implémentation des différentes phases du paradigme MapReduce selon nos besoins. Nous présenterons des pseudo-codes pour l'ensemble des phases accompagnés d'explication pour bien saisir les nuances dans la logique. Nous terminerons par présenter un exemple complet émulant le traitement des différentes phases.

Nous continuerons par présenter l'implantation de l'algorithme dans les différents environnements MapReduce. C'est-à-dire que nous présenterons l'ensemble des objets nécessaires permettant de donner vie à notre algorithme d'analyse. Sans oublier de présenter les particularités propres et incontournables que nécessite chaque environnement.

Par la suite, nous présenterons les possibilités existantes pour créer un cluster MapReduce. Soit dans deux univers différents, dans deux configurations distinctes.

Finalement, nous conclurons en présentant des tests de performance effectués dans nos environnements de références. Pour ce faire, nous présenterons les propriétés étudiées et les sources de données analysées.

REMERCIEMENTS

J'aimerais prendre l'occasion qui m'est offerte pour remercier l'ensemble des personnes qui ont eu un impact positif pour ma maîtrise. Du plus petit remerciement au plus grand. Autrement dit, du plus petit impact au plus grand.

En premier lieu, j'aimerais remercier mademoiselle Annie LeBel, pour avoir intégré le mot «maîtrise» dans mon univers universitaire. La raison est que c'est elle qui a intégré ce qu'est une maîtrise dans mon esprit, ce qui à d'intéressant à en faire une et me familiariser avec le processus d'inscription. Cela m'a donc permis de voir l'utilité d'une telle démarche et m'a certainement aidé à avoir le goût d'effectuer ma maîtrise.

Par la suite, j'aimerais remercier mon professeur, monsieur Sylvain Hallé, pour la façon qu'il a interagi avec moi durant mon cheminement académique. Ce dernier, lorsque j'étais en processus de questionnement à savoir si oui ou non j'allais effectuer une maîtrise et à pris le temps de bien définir ce que j'étais. Autrement dit, il a pris le temps de trouver un sujet pour lequel j'avais un vouloir d'en apprendre plus, soit le «parallélisme» et l'interaction des données. Pour me donner un avant-goût, il m'a donné l'opportunité à ma dernière session de mon baccalauréat d'effectuer un cours spécial dans son univers de recherche. Par la suite, lorsque j'ai pris ma décision finale, il m'a accueilli et accepté en tant qu'étudiant de maîtrise. Ensuite, j'ai beaucoup apprécié la proposition de maîtrise qui m'a fournie et je l'ai effectuée en tant que ma maîtrise.

Finalement, j'aimerais remercier mes parents, Lyna Soucy et Marc-André Boivin, pour leurs supports et leurs aides durant mes études. Ils ont toujours su m'aider tout en me permettant de faire mes propres choix. J'aimerais les remercier d'être d'accord sur le fait que des études universitaires ne sont pas superflues. Ils m'ont encouragé à faire un baccalauréat en informatique et lorsque j'ai décidé d'aller en maîtrise, ils n'ont jamais trouvé que c'était une perte de temps. De plus, ils ont continué à m'aider à être fiers de moi.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE 1

INTRODUCTION

Depuis la création du premier ordinateur, l'informaticien eut besoin de mécanismes permettant de s'assurer du bon fonctionnement de l'ordinateur. C'est alors qu'une question incontournable c'est posé : mais comment vérifier ce fait ? Pour pouvoir déterminer s'il fonctionne correctement, il faut s'assurer que les applications ou systèmes de ce dernier font l'entièreté de ce pour quoi ils ont été créés. Pour ce faire, il faut avoir de l'information sur leurs exécutions ou traitements. Ces informations sont appelées des *événements* que l'on peut sauvegarder sous forme séquentielle, selon leur ordre d'apparition, dans des journaux ou traces. Il est important de savoir qu'il n'existe pas qu'un seul type de systèmes qui peut produire des traces d'événements, ce qu'on appelle également des logs, lors de leur exécution. Par exemple, un système de gestion des achats peut produire un événement pour chaque facture créée, chaque modification des informations d'un dossier et chaque paiement effectué. Un site de clavardage peut enregistrer un événement à chaque commentaire écrit par un utilisateur. De même un système de sécurité peut émettre un événement à chaque modification effectuée dans le système qu'il gère. Il est même possible de voir un jeu vidéo comme un système qui émet des événements, par exemple pour chaque action effectuée par le joueur.

L'usage qu'on peut faire de ces événements n'est pas toujours le même; il dépend du contexte et des besoins de l'application. Par exemple, pour un système de gestion des

achats, cela permettrait de déterminer si toutes les factures ont été payées, valider le nombre d'articles en inventaire et découvrir l'article qui reste le moins longtemps en inventaire. Dans un site de clavardage, cela permet de déterminer les sujets les plus populaires, de mieux cibler les goûts de leurs utilisateurs pour vendre leurs espaces publicitaires et de déterminer les prochaines modifications à apporter au niveau de l'utilisation du site. Pour un système de sécurité, cela permettrait de pouvoir s'assurer de qui a fait quoi et de déterminer si un utilisateur essaie de s'approprier des droits supérieurs au sien. Pour un jeu vidéo, cela permettrait de s'assurer qu'un utilisateur n'effectue pas des opérations illégales et que le jeu vidéo ne permet pas des manipulations illogiques.

Une question s'impose, mais comment fait-on pour déterminer ces faits à travers une trace d'événements ? C'est à ce moment que le principe d'analyse d'événements devient utile. On doit tout d'abord déterminer tout ce que l'on veut observer dans les logs. Ensuite, on utilisera un langage formel pour pouvoir l'exprimer. Par la suite, on doit trouver la façon nous permettant de trouver ces comportements dans les logs. C'est alors que l'on doit utiliser un algorithme qui interprète les expressions du langage et qui permet ensuite d'essayer de rechercher la présence des comportements dans les logs.

Un phénomène non négligeable a été observé dès le début de l'utilisation de ce principe : la complexité des systèmes ne cesse d'augmenter. Ceci fait en sorte que les systèmes émettent de plus en plus de logs à analyser. Par exemple, au niveau des systèmes de gestion des achats, les internautes commandent de plus en plus leurs biens de consommation en ligne. Dans les faits, entre 2008 et 2013, les Canadiens ont doublé leurs

achats en ligne[1]. Il s'agit d'une tendance internationale qui obligera tous les systèmes de gestions des achats à produire plus d'événements pour être capable de continuer à suivre leurs exécutions.

Depuis les dernières années, il est possible de constater que la puissance en GHz des ordinateurs n'augmente plus. Ce phénomène est causé par l'incapacité à augmenter la fréquence d'horloge sans dissipation d'énergie croissante [12]. Par conséquent, c'est plutôt le nombre de cœurs, « cores », qui augmente. Cette conséquence change alors la façon d'utiliser les cœurs et mémoires de l'ordinateur. Un tel ensemble de ressources permet de décomposer le traitement de façon différente. La structure multicœur permet alors d'effectuer des tâches en simultané, elles sont dites *parallèles*.

Il est cependant important de noter que, pour certains problèmes, cette division n'est pas appropriée; par exemple, pour des tâches ayant une dépendance directe avec la ou les autres tâches, ou pour une tâche effectuant un traitement minime. Toutefois, on doit savoir qu'il est toujours possible d'effectuer ces traitements en parallèle. Dans les exemples ci-dessous, le premier permet d'illustrer une dépendance directe entre deux additions. Dans ce cas ci, l'usage du parallélisme n'est pas appropriée, puisque le temps que prendrais la première tâche pour effectuer le traitement de la deuxième est au moins égale ou sinon inférieur au temps de total de la deuxième tâche. La raison est que la deuxième tâche, même si créée et initialisé en même temps que la première attend la fin de celle-ci pour effectuer le traitement total.

Dans l'exemple du traitement minime, on peut voir que diviser le traitement est totalement inutile. La raison est que le temps passé à initialiser les sous processus et leur

[1] http://www.rcinet.ca/fr/2013/06/14/les-canadiens-doublent-leurs-achats-en-ligne-en-cinq-ans/

faire effectuer le traitement est supérieur au temps total que prendrais une tâche à effectuer le traitement total de l'addition.

Dépendance directe :

Tâche 1 : 1500 + 2812 + 5698 + 2367 + 59667 = 72044

Tâche 2 : Résultat Tâche 1 + 5698 + 5536 = 83278

Traitement minime :

Tâche 3 : 2 + 3 + 4 + 5 = 14

Sous-Tâche 3.1 : 2 + 3 = 5

Sous-Tâche 3.2 : 4 + 5 = 9

Sous-Tâche 3.3 : Résultat Sous-Tâche 1 + Résultat Sous-Tâche 2 = 14

En somme, la raison pour laquelle la division n'est pas appropriée pour toutes les tâches est que cela apporte un gain de performance minime voir inexistant. Ainsi, le traitement de la tâche en mode séquentiel prend soit le même ou même un peu moins de temps.

Pour continuer à obtenir des gains au niveau de la performance, il faut trouver de nouvelles manières de tirer parti de la possibilité de paralléliser les tâches, en les réécrivant de manière à faciliter leur distribution sur plusieurs nœuds. Pour pouvoir créer des tâches parallèles efficaces, il faut tout d'abord déterminer l'environnement de travail et ses outils. Soit on utilise les langages de programmation parallèles permettant de créer une tâche dite parallèle, ou alors on utilise des environnements de travail permettant d'effectuer une tâche sur un grand nombre de nœuds. Dans tous les cas, le choix des outils est crucial, puisqu'il implique une différence entre la structure des tâches. Une tâche parallèle n'a pas pour but d'êtres reproduite en un nombre X. Dans les faits, son

nombre de tâches est soit fixe dû aux ressources disponibles de l'ordinateur ou selon ses besoins au niveau du traitement à effectuer. Pour les environnements de travail, ils peuvent être un regroupement d'ordinateurs et être reliés par un réseau local ou reliés via la technologie de « Cloud computing ». Donc, cela apporte comme gain plus de cœurs pour effectuer la même tâche et une toute autre gestion des données. Il est également important de savoir qu'un tel environnement est distribué et scalable, ce qui est idéal pour l'analyse de tous les grands ensembles de données, par exemple de log de serveurs. Toutefois, un de ses avantages le plus importants est qu'il permet de lancer un nombre variable d'unités de traitement selon les besoins de la tâche. Donc, si nous avons besoin d'effectuer le même traitement sur deux ensembles de données et que l'un d'eux est le double de l'autre, alors l'environnement nous permettra d'utiliser plus de nœuds pour effectuer le traitement. Le but est de bien sûr augmenter la performance de traitement selon les ressources disponibles. Tandis que pour les tâches parallèles, le nombre d'unités de traitement restera le même et cela aura pour impact de prendre plus de temps pour effectuer le même traitement. Un des environnements le plus reconnu et le plus utilisé dans le domaine de la gestion de traitement parallèle de grands ensembles de données est l'environnement Hadoop MapReduce[2].

La plupart des outils qui existent présentement implémentent les différents algorithmes d'analyse de logs de serveurs sans prendre avantage des capacités du parallélisme. Il est important de noter que quelqu-uns d'entre eux montrent une certaine ouverture au parallélisme. Pourtant, après une étude en profondeur, on peut s'apercevoir que les phases d'analyses sont divisées en sous-tâches, mais exécutées une après l'autre.

[2] https://hadoop.apache.org/

Dans les faits, on peut en conclure que les algorithmes existants d'analyse de logs n'utilisent pas du tout le principe du parallélisme.

Dans ce mémoire, on s'attarde à un cas bien précis, soit la vérification de propriétés de la logique *Linear Temporal Logic (LTL)* sur des logs en format XML en utilisant le paradigme de programmation MapReduce. Pour ce faire, on implémente et expérimente un algorithme d'analyse permettant d'effectuer ce traitement de manière parallèle et distribuée. Au chapitre 2, une revue de la littérature du domaine nous permettra de comprendre de ce qu'est la LTL, l'environnement Hadoop et les outils existants. Au chapitre 3, le paradigme MapReduce sera décrit et expliqué au moyen d'un exemple. Ensuite, le chapitre 4 présentera un algorithme de vérification de propriétés LTL appliqué à MapReduce.

Le chapitre 5 présente MrSim, une implémentation *monothread* du paradigme MapReduce écrite en Java, et sur laquelle a d'abord été testé l'algorithme que nous présentons. Par la suite, le chapitre 6 permet de définir les modifications techniques que j'ai apportées pour que notre algorithme d'analyse de logs, que l'on appelle LTLValidator, puisse cette fois fonctionner dans l'environnement Hadoop MapReduce. Le chapitre suivant décrit comment j'ai mis en place un cluster Hadoop MapReduce. Enfin, le chapitre 8 présente les tests de performance entre les différentes implémentations de l'algorithme.

La dernière section est la conclusion, qui permet de bien saisir les faits qui sont ressortis à la suite de la recherche qui a mené à ce mémoire. Nous avons en effet pu observer plusieurs faits intéressants. L'observation la plus importante est qu'il est possible de créer un algorithme d'analyse de logs dans un environnement Hadoop

MapReduce dont la vitesse d'exécution rivalise avec celles des outils d'analyse de trace non parallèles. Toutefois, Hadoop impose des limitations d'ordre technique qui ne sont pas inhérentes à MapReduce et qui ont demandé plusieurs modifications à l'implémentation. De plus, il produit un nombre gigantesque d'éléments de données, appelés *tuples*, pour des formules de taille moyenne. Lorsque les formules sont de grandes taille, cette réalité fait planter notre environnement de travail de référence, puisqu'il manque d'espace mémoire pour traiter l'ensemble des tuples générés pour analyser la formule.

Finalement, il est important de noter que les résultats de la recherche décrit dans ce mémoire, on fait l'objet d'une publication : Benjamin Barre, M.K., Maxime Soucy-Boivin, Pierre-Antoine Olivier and Sylvain Hallé, *MapReduce for Parallel Trace Validation of LTL Properties*, in *RV 2012*. 2012: Istanbul, Turquie. Springer : Lecture Notes in Computer Science 7687, 184-198.

Celle-ci a remporté le prix du « Best Paper Award » lors de la conférence « 3rd International Conference on Runtime Verification (RV 2012)[3]».

Une version étendue a été publiée au *Jounal of Cloud Computing* (Springer) en avril 2015: *Journal of Cloud Computing: Advances, Systems and Applications.2015, 4:8*

DOI: 10.1186/s13677-015-0032-x .

[3] http://uqactualite.uqac.ca/sylvain-halle-et-quatre-etudiants-remportent-un-prix-international/

CHAPITRE 2

REVUE DE LITTÉRATURE

Nous allons nous intéresser à la problématique d'analyse de traces. Il est important de se rappeler qu'il n'y a pas de sources uniques pour les événements. Dans les faits, tout système peut en devenir une, tels que des applications de télécommunication, des systèmes d'exploitation, des services web et même des fusées qui vont sur la lune. On peut alors en conclure qu'il n'y a que le talent des informaticiens qui est la limite pour créer des applications qui génèrent des traces. Il est également important de savoir que les traces n'ont pas de format obligatoire. Dans les faits, on peut presque dire que chaque application d'analyse possède son propre format; il n'y a que quelques applications qui partagent strictement le même format d'écriture en entrée. Toutefois, le but de l'ensemble des traces est le même, sauvegarder de l'information sur la logique du traitement effectué par le système, quel qu'il soit. Ces traces nous permettent ensuite de déterminer les actions exécutées antérieurement et d'en tirer des faits et conclusions sur le bon fonctionnement du système. De plus, ils nous permettent de déduire des actions futures qui doivent ou peuvent arriver au niveau du système. Sans oublier qu'ils nous permettent de trouver les moments où la logique de notre système a été altérée par un bogue et/ou intrusion.

Cette section dresse un panorama du problème de l'analyse des logs. On retrouvera dans la première sous-section une présentation de la logique temporelle linéaire (LTL)

utilisée au cours de cette recherche. Nous nous attarderons surtout sur les opérateurs de cette logique et ce que l'on peut exprimer avec ceux-ci. Dans la deuxième sous-section, on trouvera une présentation des différents outils d'analyse de logs séquentiels existants. Nous aborderons le langage qu'ils emploient ainsi que leurs différentes applications. Dans la dernière section, nous allons finalement discuter des origines et de ce qu'est l'analyse parallèle.

2.2 La logique temporelle linéaire (LTL)

2.2.1 Trace d'événements

Une trace est un regroupement d'événements avec une structure uniforme, prédéterminée selon les besoins du système qui les émet et qui contient les informations que le système a choisi de transmettre. Dans le présent document nous utiliserons la notation m^i qui veut dire : la trace m prise à partir de son i-ème événement. La structure de la trace d'événements que nous utilisons est de type XML, puisque les normes du XML deviennent un atout pour la logique de notre trace. L'avantage qui nous intéresse le plus est le fait que les données sont toujours contenues entre deux *balises*, l'une appelée ouvrante et notée <balise>, et une fermante notée </balise>. Cette notation nous permet alors de bien structurer les événements à étudier, leurs paramètres et leurs valeurs. Tous les éléments que nous étudions sont également contenus dans une unique balise de type racine, <Trace></Trace>. Celle-ci contiendra un ensemble de balises secondaires, soit dans notre cas les balises d'événements, représentées par <Event>...</Event>. Elles ont pour but de représenter un événement émis par son système source. Ces balises contiennent elles-

mêmes des variables représentes par des sous-balises, de format <p « nombre »>… </p « nombre »> qui à l'intérieur contiennent des valeurs numériques.

Ce format de trace XML particulier est représenté dans la figure 1. Nous le définissions comme traditionnel, puisque ce dernier est le format que nous avons utilisé au début de nos recherches. De plus, il est important de noter qu'un autre format aurait pu nous permettre de contenir les mêmes informations et de nous permettre de pouvoir effectuer une analyse avec succès. Toutefois, ce format répondait le mieux à nos besoins.

```
1  <Trace>
2
3  <Event>
4   <p2>21</p2>
5   <p0>0</p0>
6   </Event>
7
8  <Event>
9   <p4>6</p4>
10  <p2>21</p2>
11  <p0>0</p0>
12  </Event>
13
14 <Event>
15  <p4>2</p4>
16  <p0>0</p0>
17  </Event>
18
19 </Trace>
```

Figure 1 : Trace XML traditionnelle

2.2.2 LTL

La LTL a été introduite par Pnueli en 1977 [14]. Cette dernière permet d'énoncer des propriétés sur des systèmes concurrents et réactifs. Dans le cadre de nos recherches,

c'est ce langage que nous utiliserons pour exprimer des propriétés sur des traces d'événements. Les variables propositionnelles constituent les éléments de base de la LTL, telles que p, q, …, qui servent à exprimer des conditions booléennes (i.e. vraies ou fausses) sur les événements individuels du log.

Pour les besoins de ce mémoire, un événement de la trace dénotée m_0m_1…, est représenté par m et ce pour une séquence d'événement durant une période de temps. Chaque événement est une entité individuelle faite de plus d'un couple. Un couple est un paramètre et une valeur associée ensemble, avec des noms et types arbitraires. Il est important de noter que le schéma, le nombre et le nom de chaque paramètre de chaque événement, sont des éléments que l'on ne connaît pas à l'avance et qu'il n'a pas nécessairement de lien entre eux. Dans les faits, chaque événement pourrait posséder sa propre structure, si tel est le besoin.

Pour simplifier les traitements et l'écriture des formules, nous utilisons dans le cadre de cette recherche, la notation XPath pour définir les propriétés à étudier. Cette notation nous permet de toujours associer un paramètre avec sa valeur recherchée. Voici quelques exemples :

Exemple 1 : affirmer que la valeur de x dans l'événement actuel est égale à 0 s'écrira {x/0}.

Exemple 2 : affirmer que la valeur de p6 dans l'événement actuel est égale à 9 s'écrira {p6/9}.

Comme nous venons juste de voir, la notation de Xpath, permet de définir des propriétés. Toutefois, il nous est impossible, jusqu'à maintenant d'élaborer des observations complexes. C'est alors que les connecteurs propositionnels rentrent en jeux. Ces derniers nous permettent d'associer plusieurs contraintes et faits. Il y a le connecteur → qui est une implication et qui signifie « si … alors … ». Donc, cela veut dire que si on peut observer la propriété de gauche, cela impliquera que la propriété de droite doit l'être également. Ensuite, il y a le connecteur ←→ qui signifie « …si et seulement si… » et qui implique que les propriétés de gauche et de droite doivent être observées toutes les deux, ou pas du tout, pour être considérée comme étant respectée. Il y a aussi le connecteur ∧ qui signifie « et » et qui implique que la propriété de gauche doit être observée et ainsi que celle de droite doit l'être aussi, pour être considérée comme respectés. Par la suite, il y a le connecteur ∨ qui signifie « ou » et qui implique que la propriété de gauche ou de droite doit être observée pour être considérée comme étant respectée. Finalement, il y a le connecteur ¬ qui représente la négation d'un fait. Cela veut dire que la propriété à laquelle ce connecteur est associé ne doit pas être observable pour être considérée comme étant respectée. Voici des exemples :

Exemple 1 : Lorsque la propriété p2 est égale à 2, on doit observer que la propriété 4 est égale à 4 : {p2/2}→{p4/4} .

Exemple 2 : La valeur de la variable p5 est égale à 10 uniquement quand p6 est égal 6 et inversement : {p5/10} ←→{p6/6} ou {p6/6} ←→{p5/10}.

Exemple 3 : Lorsque la variable p3 est égale à 3 et que p5 est égale à 5, cela implique que le fait est respecté : {p3/3} ∧{p5/5}.

Exemple 4 : Pour que le fait soit respecté, il faut que la variable p6 soit égale à la valeur 6 ou 8 : {p6/6} ∨ {p6/8}.

Exemple 5 : Lorsque la variable p2 est égale à 4, cela implique que le fait à observer n'est pas respecté. Donc, la variable p2 peut prendre n'importe quelle valeur sauf la valeur 4 : ¬ {p2/4} .

2.2.3 Opérateurs Temporels

Les opérateurs temporels permettent de pousser plus loin la façon d'écrire une formule, en permettant d'énoncer des faits sur la manière avec laquelle les événements peuvent se succéder dans une trace. Les prochaines sections présentent chacun des opérateurs.

2.2.3.1 Opérateur **G**

L'opérateur **G** signifie « globally », la formule **G**φ que φ est vraie dans tous les événements de la trace, à partir de l'événement en cours. C'est-à-dire que la propriété à observer est toujours vraie ; voici des exemples de telles formules:

Exemple 1 :**G**({p1/1}) signifie intuitivement que la valeur du paramètre p1 doit être égale à 1 dans tous les événements de la trace.

Exemple 2 : **G**({p2/2} →{p4/4}) cela implique que lorsque la valeur de la variable p2 est égale à 2, la variable p4 sera égale à 4, et ce, dans tous les événements de la trace .

2.2.3.2 Opérateur **F**

L'opérateur **F** signifie « eventually », la formule **F** φ est vraie si φ est vraie dans un événement futur de la trace. Autrement dit, cela veut dire que la propriété sera respectée par un événement futur contenu dans la trace, à partir de l'événement courant. Exemples :

Exemple 1 : **F** {p5/6} signifie qu'éventuellement la variable p5 sera égale à 6 dans un événement futur de la trace.

Exemple 2 : **F**({p8/8}←→{p7/17}) signifie qu'éventuellement la variable p8 sera égale à 8 et que p7 sera égale à 17, et inversement, dans un ou des événements futurs de la trace.

2.2.3.3 Opérateur **U**

L'opérateur **U** signifie « until », la formule φ **U** ψ est vraie si φ est vrai pour tous les événements jusqu'à ce qu'un événement vérifie ψ. C'est-à-dire que la propriété de gauche doit être observée pour l'ensemble des événements jusqu'à ce que la propriété de droite le soit. Voici des exemples de formules utilisant cet opérateur.

Exemple 1 : {p9/9} **U** {p1/1} signifie que dès que la variable p9 égale à 9 a été observée, l'ensemble des variables suivantes doit être pareilles à cette dernière jusqu'à ce que la variable p1 est égale à 1 soit observée, pour être considéré comme étant respectée. Toutefois si on observe dès le départ la variable p1 égale à 1, la formule est quand même respectée, puisqu'il n'existe aucune variable p9 avant la variable p1.

Exemple 2 : ({p2/2} →{p4/4}) **U** {p5/6} cela implique que lorsque p2 est égale 2, que p4 est égale à 4 dans tous les événements de la trace, et ce, jusqu'à ce que p5 soit égale à 6, pour que la propriété soit considérée comme étant respectés.

2.2.3.4 Opérateur **X**

L'opérateur **X** signifie « neXt » et est vrai chaque fois que φ est vrai dans le prochain événement de la trace. Donc, cet opérateur permet d'établir une relation entre l'événement courant et le suivant. Exemples :

Exemple 1 : **X**{p3/3} signifie que la prochaine variable à observer est p3 et que cette dernière est égale à 3, pour que cette dernière soit considérée comme étant respectée.

Exemple 2 : {p2/2}→**X**{p4/4} cela implique que lorsque la variable p2 est égale 2 et que la prochaine variable est p4 avec comme valeur 4, pour que la propriété soit respectée.

2.2.4 Syntaxe et Sémantique formelle de la LTL

Formellement, on peut définir la syntaxe d'une expression LTL comme suit

Définition 1 (Syntaxe) :

1) Si x et y sont des variables ou des constantes, alors x = y est une formule **LTL** ;

2) Si φ et ψ sont des formules **LTL**, alors ¬ φ, φ ∧ ψ, φ ∨ ψ, φ → ψ, **G**φ, **F**φ, **X**φ, φ **U** ψ sont également des formules **LTL**

La ***sémantique*** de chacun de ces symboles décrit comment celui-ci doit être interprété lorsqu'une formule est évaluée sur une trace, comme suit :

Définition 2 (Sémantique) :

Nous disons qu'une trace de messages *m* satisfait la formule *LTL* φ, que nous écrivons m |= φ, si et seulement si elle respecte les règles du Tableau 1. Nous définitions la sémantique des autres connecteurs avec les identités suivantes : $\varphi \wedge \psi \equiv \neg (\neg \varphi \vee \neg \psi)$, $\varphi \rightarrow \psi \equiv \neg \varphi \vee \psi$, $\mathbf{G}\varphi \equiv \neg (\mathbf{F} \neg\varphi)$

$m \models c1 = c2 \leftrightarrow$ c1 est égal à c2

$m \models \neg \varphi \leftrightarrow m \not\models \varphi$

$m \models \varphi \vee \psi \leftrightarrow m \models \varphi$ ou $m \models \psi$

$m \models \mathbf{F}\, \varphi \leftrightarrow m^i \models \varphi$ pour un $i \geq 1$

$m \models \mathbf{X}\, \varphi \leftrightarrow m^2 \models \varphi$

$m \models \varphi\ \mathbf{U}\ \psi \leftrightarrow m^j \models \psi$ pour un j et $m^i \models \varphi$ pour $i < j$.

Tableau 1 : Sémantique pour la *LTL*

2.3 Les outils séquentiels

Nous allons présenter dans cette section une liste d'outils d'analyse de logs séquentiels qui figurent parmi les plus utilisés. Il est important de noter qu'ils existent d'autres outils pour effectuer l'analyse de logs. Toutefois, voici la liste de ceux que l'on connaît et qui sont parmi les plus utilisés. Nous allons décrire leur but, les éléments qu'ils utilisent et nous allons aborder leur utilité.

2.3.1 BeepBeep

BeepBeep [1] est un moniteur d'exécution permettant de recevoir un flux d'événement et par la suite il peut les analyser sur des formules LTL qui lui ont été soumises. De plus, il est également un applet Java permettant de fortifier l'exécution des clients JavaScript et également les serveurs Java. Puisqu'il est facile pour cet outil de ramasser les événements qui sont créés par une source et de les analyser sur les données voulues. Il peut également fonctionner en différé, puisqu'il est possible d'effectuer une analyse d'événement à partir d'un fichier avec une formule voulue. Ce dernier est un logiciel libre et disponible gratuitement. Pour effectuer l'analyse d'événements, BeepBeep utilise un langage dérivé de la LTL, le LTL-FO+, pour pouvoir implémenter sa logique d'analyse.

2.3.2 Logscope

Logscope [2] est un outil dit général dans l'analyse de logs. Il a été tout d'abord créé et écrit pour aider les ingénieurs de test à comprendre comment un système se comporte. L'outil a été développé dans la cadre d'une mission spatiale de la NASA et a reçu un prix pour avoir amélioré la qualité et la productivité de l'application MSL Flight. Toutefois, il est important de savoir que celui-ci peut s'adapter à tout système de journalisation et donc, à tout format de logs généré par un système. Une de ses forces, c'est qu'il permet de tester et d'analyser l'interaction des différents sous-systèmes entre eux. Pour ce faire, il utilise son propre langage, le Logscope, permettant d'effectuer l'analyse

des logs. Il faut bien sûr implémenter la logique que l'on veut établir, utiliser et tester à l'intérieur de ce dernier.

2.3.3 Maude

Maude [3] utilise comme langage la logique temporelle linéaire pour pouvoir effectuer l'analyse des logs soumis. Ce dernier est un outil efficace permettant d'effectuer un très grand nombre de réécritures par seconde. De plus, grâce à ses caractéristiques de métalangage, il se révèle un outil de grande qualité pour pouvoir créer des environnements exécutables pour différentes logiques, démonstrateurs, modèles de calcul et même des langages de programmation. Il est également un moteur de surveillance d'événements intéressant pour effectuer de l'analyse de logs. Il soutient aussi la modularisation dans le style OBJ[3] .

2.3.4 Monid

Monid [4] est un environnement de travail créé pour pouvoir effectuer de la détection d'intrusion basée sur le *runtime monitoring* avec des spécifications de logique temporelle. Monid utilise son propre langage de spécification, EAGLE. Le langage permet de rendre accessible la spécification de comportements de sécurité d'un système et cela les rend donc accessibles à l'analyse. De plus, Monid permet de spécifier les scénarios d'intrusion dans un format de formule expressif et efficient. Il est important de noter que l'application effectue une analyse dite *en ligne*, ce qui veut dire que les événements sont analysés dès que l'application les reçoit. Un exemple concret au niveau de l'utilisation de Monid est l'analyse d'une attaque par essai de mot de passe. La complexité de cette attaque

est de déterminer les accès véridiques et les accès frauduleux. La façon dont l'application s'en sort est d'effectuer une collecte de statistiques temporelles à l'exécution et de faire une proposition sur la base de ces dernières.

2.3.5 MonPoly

MonPoly [5] est un outil d'analyse utilisant un langage dérivé de la LTL, appelé MFOTL. Ce langage permet de spécifier des politiques de sécurité complexes et réaliste. Dans les faits, cet outil permet de gérer et faire le suivi des politiques systèmes à travers d'événements. On peut alors en déduire que c'est un outil orienté vers l'univers de la sécurité. De plus, il est important de savoir que la classe des politiques couvertes constitue un ensemble des propriétés de sécurité. On peut tester et confirmer la conformité des propriétés par le suivi des traces du système.

2.3.6 ProM

ProM [7] est un outil *open-source* permettant d'effectuer l'analyse de logs. Ce dernier a été créé dans le but de pouvoir extraire des événements, de format MXML, à partir des informations systèmes. L'outil contient des plug-ins spécifiques, qui sont des recherches effectuées sur l'analyse de certains types de logs dans une logique donnée. Il est alors important de savoir qu'il existe des plug-ins permettant d'analyser différents systèmes et structures de données. Donc, les chances qu'un utilisateur doit en écrire un nouveau en Java sont faibles. Toutefois, si telle est le cas, il existe XESame qui fournit un moyen générique pour effectuer l'extraction d'événements à partir d'un journal selon certaines données. Cela implique alors que la source de données est conçue dans un format

prédéfinie. Selon Verbeek et al [7], ProM dispose de 286 plug-ins dans sa version 5.2. Au niveau du traitement de la logique interne, le langage utilisé est la LTL.

2.3.7 RuleR

RuleR [8] est un outil basé sur des règles primitives conditionnelles, qui est un algorithme simple et facile permettant de vérifier en temps réel de façon efficace des événements contenus dans des logs. De plus, il est possible de transformer un automate non déterministe en une règle que l'outil sera capable d'analyser. L'avantage se situe au niveau du fait, qu'un automate peut représenter la logique des étapes d'un événement selon la valeur qui est contenue à la prochaine étape. De plus, l'outil contient plusieurs règles prédéfinies ce qui permet de simplifier l'analyse des logs. Il est important de noter que RuleR utilise le langage RuleR, inspiré d'Eagle, pour représenter ses règles et ce dernier est basé sur la LTL.

2.3.8 Saxon

Saxon [9] est un moteur XQuery permettant d'effectuer des analyses de logs de façon séquentielle. Ce dernier permet d'étudier tout type de logs de format XML. Il est important de noter que le XML est un langage adaptatif tout en ayant des normes au niveau de sa structure. Ce qui est pratique, puisqu'il est possible de traduire des équations LTL en XML de façon simple et rapide. La logique que ce dernier peut utiliser est la LTL-FO+ au niveau des éléments à analyser. Saxon est un moteur mature, libre d'utilisation et est de niveau industriel.

2.3.9 SEQ. OPEN

SEQ. OPEN [10] est un compilateur *Open/Cæsar-compliant* qui prend un fichier de séquence, comme des logs, et le transforme en un fichier pour que l'API puisse le traiter. Ce dernier est un ensemble de *n* traces contenues dans un fichier de format Seq, qui peut être considéré comme un système de transition étiqueté avec trois types d'états. Il y a l'état sans issues (derniers états, avec 0 successeur); l'état normal (états intermédiaires, avec 1 successeur); et l'état initial (commun à toutes les traces, avec *n* successeurs). Pour pouvoir être capable de traiter la logique voulue, il utilise le langage μ-calculus qui permet de représenter les propriétés d'étude.

2.3.10 Résumé

On retrouve dans le tableau 2 une liste récapitulative des outils d'analyse de logs séquentiels présentés précédemment. Plus précisément en ordre de colonne, on peut retrouver une liste d'outils séquentiels, le langage que ces derniers utilisent et si oui ou non l'outil est accessible pour l'utilisation immédiate.

Tableau 2 : Liste sommaire d'outils séquentiels

Outils	Langage	Disponible ?
BeepBeep [1]	LTL-FO+	Oui
Logscope [2]	Logscope	n/d
Maude [3]	LTL	Oui
Monid [4]	EAGLE	n/d
MonPoly [5]	MFOTL	Oui
ProM [7]	LTL	Oui

RuleR [8]	RuleR	n/d
Saxon [9]	XQuery	Oui
SEQ. OPEN [10]	μ-calculus	Oui

En somme, il est important de remarquer que ces outils utilisent des principes et ressources de façon intéressante. Il démontre également l'utilité diversifiée de ce genre d'algorithme et d'outils. Il nous prouve également que l'on peut suivre n'importe quelle application, pourvue que celle-ci émette des données dans un format uniforme et accessible pour l'analyse. Toutefois, ce sont des outils qui sont strictement séquentiels et aucun de ces derniers ne tient compte du parallélisme d'aucune façon.

CHAPITRE 3

MAPREDUCE

Le principe du parallélisme est né lorsque les informaticiens eurent besoin d'effectuer un grand nombre de tâches simultanément. Ils remarquèrent également qu'il est possible que l'ensemble des ressources systèmes ne soient pas utilisé en tout temps. C'est alors que la recherche commença pour élaborer des structures et/ou méthodes de travaillent permettant de maximiser l'utilisation des ressources. Les informaticiens ont alors créé les threads, des entités permettant d'effectuer du traitement en même temps que le processus principal. Toutefois, ils se sont aperçus rapidement qu'ils ont créé un nouveau problème, la synchronisation de l'exécution.

La raison est que rien, à prime à bord, ne nous permet de déterminer qu'un thread aura fini son exécution au moment ou le processus principal aura besoin des résultats de ce dernier. C'est alors qu'ils créent des outils permettant de gérer la synchronisation des informations et l'exécution d'un thread à la fois dans une section de code. Toutefois, ces éléments et/ou outils deviennent de plus en plus restrictifs avec l'avènement de l'augmentation du nombre de cœurs et de la mémoire vive.

C'est alors, que les informaticiens ouvrir les portes et lancèrent les bases de la deuxième génération d'idée sur le parallélisme, les langages parallèles. Ces derniers sont

pensés pour créer des unités de traitements spécialisés qui peuvent s'exécuter de façon indépendante, et ce, sans nombre d'unités de tâches minimum et maximum. C'est-à-dire que le nombre total d'unités varient selon les besoins de l'exécution et/ou de l'ordinateur. Ils ont alors ouvert la possibilité d'effectuer du traitement parallèle de façon plus simple, performante et efficiente. Cela a donc permis de s'attaquer de meilleure façon à des problèmes complexes et qui prenaient beaucoup de temps à résoudre.

Un de ces problèmes est l'analyse de logs de serveur, puisque la majorité des outils offerts sont munis d'algorithmes séquentiels. Les langages parallèles permettent de maximiser l'utilisation des ressources au niveau de cette problématique. Toutefois, le problème s'est complexifié au lieu de se simplifier. La raison est que les langages ne sont pas, à prime à bord, conçus pour le traitement de grand ensemble de données. Parce que ce type de traitement exige une gestion continue à savoir qui effectue quel traitement et qui reçoit les résultats de l'exécution. C'est pour répondre à cette problématique que le principe d'environnement de traitement est né. L'un des plus connues est l'environnement de traitement parallèle Hadoop MapReduce.

Depuis quelques années, l'émergence du concept de « Cloud computing » a mené à la création et publication d'une variété d'environnements informatiques. Une composante intéressante et notable de ces derniers est le MapReduce, un cadre de programmation mis en place par Google en 2004 pour le traitement d'une grande quantité de données [15]. Ce cadre est l'un des précurseurs de la soi-disante tendance «NoSQL», qui a pour but de libérer et de permettre d'utiliser autre chose que des bases de données relationnelles

classiques pour pouvoir conserver les données. Cela permet alors, une tout autre façon de développer une interaction avec les données et le traitement sur ces dernières.

Dans ce chapitre, nous allons étudier en profondeur le principe MapReduce et ce dernier est représenté dans son ensemble dans la figure 2. Les prochaines sections présenteront en détail les étapes que la figure contient.

3.1 Description des composants de MapReduce

3.1.1 L'InputReader

L'InputReader, la première étape du principe MapReduce permet de convertir les données pour qu'elles soient accessible pour le Mapper. C'est-à-dire que cette étape convertit les informations de leurs état brut, la majorité du temps contenue dans un fichier, en une série de tuples de la forme <k,v>. Ce format <*k,v*>, est utile pour contenir les informations dans l'ensemble des phases nécessaires pour effectuer le traitement. Il permet de simplifier les échanges d'informations et ce dernier se décrit comme étant un couple de clé *k* et valeur *v*. Il est important de savoir que ces éléments sont déterminés lors de la programmation de la tâche : selon la tâche à effectuer, les clés et les valeurs peuvent être de types variés. Il est également important de savoir que les données peuvent provenir de plusieurs sources, et il peut donc y avoir plus d'un InputReader qui émet des tuples.

3.1.2 Mapper

La phase Mapper survient lorsque la phase d'InputReader est terminée et reçoit les tuples de cette dernière. Il est important de noter que chacun des tuples est envoyé un à la

suite de l'autre à un mapper contenu dans l'environnement. Le mapper effectue un tri selon l'analyse voulue et, pour chaque tuple d'entrée, choisit d'émettre en sortie zéro, un ou plusieurs tuple s de la forme <*k',v'*>. De plus, il est important de savoir que cette phase n'est pas dans l'obligation de produire un tuple en sortie pour tout tuple reçu. Dans les faits, cela varie d'aucun à tant que nécessaire. Précisons que la phase d'InputReader peut être fusionnée avec la phase de Mapper. Cependant, on peut séparer le traitement des tuples entre plusieurs instances de mapper, qui seront alors toutes identiques et qui pourront s'exécuter en parallèle.

3.1.3 Shuffling

La sous-phase Shuffling survient lorsque la phase de Mapper est terminée. L'environnement reçoit l'ensemble des tuples émis par l'ensemble des mapper et effectue le shuffling. C'est-à-dire le tri des différents tuples et les regroupe selon leurs clés communes.

3.1.4 Reducer

Chaque ensemble de tuples pour une même clé *k'* est ensuite envoyé à une instance de reducer. C'est à ce niveau que l'on effectue le cœur de l'analyse voulu avec les informations. Cela implique donc que tout traitement logique voulu doit être implémenté à cette phase. Suite au traitement, cette phase émet également ses résultats dans un format de tuple <*k'', v''*>. Il est également important de savoir que la phase de Reducer n'émet pas nécessairement un tuple pour chaque tuple reçu. On peut produire d'aucun à plusieurs tuples pour un tuple reçu par le Reducer et c'est donc la logique implémentée qui fait loi.

De plus, il est important de savoir que le format des tuples, c'est-à-dire le type de clé et de valeur, n'est pas nécessairement le même que celui des tuples d'entrée. Contrairement au mapper qui reçoit les tuples un à un, sans avoir aucune hypothèse sur leur ordre, le reducer possède un accès aléatoire à l'ensemble des tuples pour une même clé.

3.1.5 OutputWriter

Finalement, c'est la phase du OutputWriter qui s'occupe de transformer les résultats dans le format voulu. Par exemple, c'est cette phase qui peut écrire, dans un format prédéterminé et implémenté dans cette dernière, les informations dans un fichier. Le but traditionnel de cette phase est de placer les éléments dans un format permettant l'utilisation des résultats par un système.

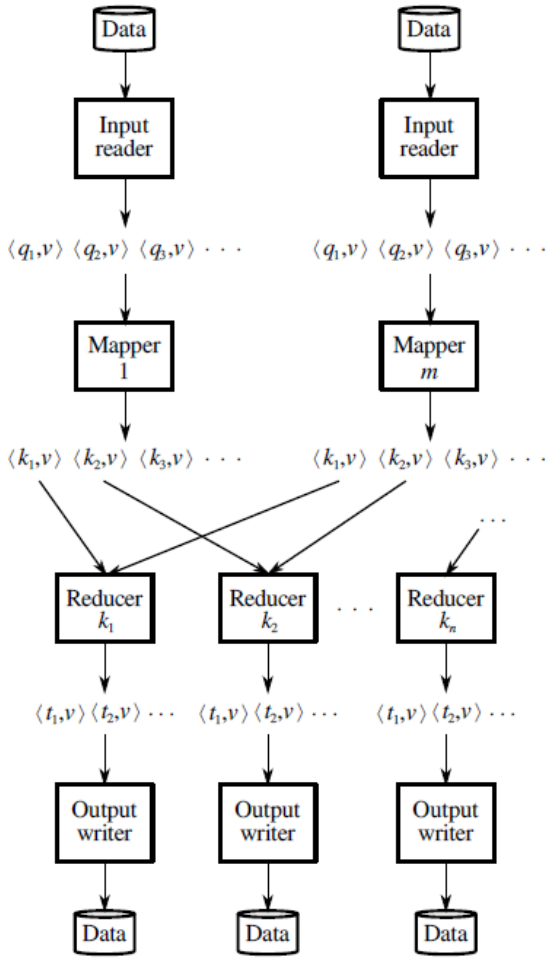


Figure 2 : Les différentes étapes de traitement de données du principe MapReduce

3.2 Exemple de tâche

Une tâche MapReduce se divise en plusieurs sous phase, comme nous venons tout juste de le voir. Voici un exemple de référence dans la littérature MapReduce, soit le WordCount, présenté en pseudo-code. La tâche a pour but de calculer le nombre de fois que chacun des mots, avec son écriture précise, a été utilisé dans un texte donné.

3.2.1 Mapper

Le Mapper, représenté dans le pseudo-code de la figure 3, reçoit les informations, soit l'ensemble des mots contenus dans le texte avec une valeur. Cette dernière est tout simplement l'endroit en octets où commence le mot dans l'entité du texte. Ce qui permet donc d'affirmer hors de tout doute que chaque nombre sera unique et cela permettra d'identifier chacun des mots de façon unique.

La fonction map implémente le traitement de la logique dont l'on a besoin pour notre problème. Dans notre cas, pour le WordCount, la phase d'InputReader prend l'ensemble des mots, les convertir en chaîne et les places un à un dans un tableau. Ensuite, la fonction map les reçoit, parcourt l'ensemble des chaînes contenues dans le tableau et les émettra en tant que tuple. Dans ce cas-ci, le mot agit comme étant une clé et est accompagné du nombre de fois qu'il est présent, soit 1. Par la suite nous utilisons le contexte, qui est lien entre la phase Mapper et l'environnement pour émettre les tuples résultats. Donc, pour que le reste du processus puisse les utiliser, il faut les écrire via le contexte.

Exemple 1:

Dans le cadre de cette explication, nous utiliserons le texte suivant :

Pomme
Orange
orange
Pomme
orange
Pomme

La phase Mapper recevra cet ensemble de données :
<0, Pomme> , <7, Orange> , <15, orange> , <23, Pomme> , <30, orange> , <38, Pomme>

Le mapper émettras les tuples suivants :

<Pomme 1>, <Orange 1>, <orange 1>, <Pomme 1>, <orange 1>,<Pomme 1>

Classe TokenizerMapper **hérite** Mapper < Objet, Texte, Texte, Nombre>

```
Fonction map (Objet clé, Texte valeur, Contexte contexte)
    itr[] =  ConvertirChaîne(valeur)
    Pour i = 0 ; i<=  itr. Taille ; i++
       contexte.écrire(itr[i],1)
```

Figure 3 : Pseudo-code pour le Mapper WordCount

3.2.2 Reducer

Le Reducer, représenté dans le pseudo-code de la figure 4, reçoit les tuples regroupés selon la clé émise. Dans ce cas-ci, les tuples ont été regroupés selon les mots. Il est important de savoir que le mot « Orange » et « orange » n'est pas considéré comme étant le même, puisque leurs valeurs en octets est différentes. Ils sont donc considérés comme étant 2 mots différents et seront donc traité dans 2 Reducers différents.

La fonction reduce permet d'effectuer le traitement voulu, soit compter le nombre de fois où le même mot apparaît dans un texte. Pour ce faire, il parcourt l'ensemble reçu à traiter et additionne le nombre associé à chacun des mots. Toutefois, il est important de savoir que le mot, soit la clé, est envoyé une seule fois et est associé avec l'ensemble des nombres. Par la suite, après avoir traité l'ensemble contenant les nombres, il émet un tuple résultat contenant la clé avec la somme précédemment calculé. Finalement, lors de la phase d'OutputWriter, elle écrit les tuples émient par la phase Reducer selon ce que le programmeur a déterminé comme écriture.

Exemple 1 : (Voici la suite de l'exemple 1 de la section Mapper précédente)

Voici les différents ensembles de tuples que recevront les différentes phases de Reducer :

Ensemble 1 : <Pomme 1> , <Pomme 1> , <Pomme 1>
Ensemble 2 : <Orange 1>
Ensemble 3 : <orange 1> , <orange 1>

Les différents Reducer émettront les tuples suivants :

Ensemble 1 : <Pomme 3>
Ensemble 2 : <Orange 1>
Ensemble 3 : <orange 2>

Classe IntSumReducer **hérite** Reducer < Texte, Nombre, Texte, Nombre>

```
Fonction reduce (Texte clé, iterateur<Nombre> valeurs, Contexte contexte)
    sum = 0;
    Pour chaque ( val : valeurs)
        sum += val.acquérir();

contexte.écrire(clé, sum)
```

Figure 4 : Pseudo-code pour le Reducer WordCount

3.2.3 Objet de la tâche

L'appellation objet de la tâche représente dans le présent document, l'élément qui permet de configurer et de lancer l'exécution de la tâche. Nous n'utilisons pas de terme précis, puisque la configuration d'une tâche varie d'un environnement à l'autre. Sans oublier que le principe MapReduce est un paradigme de traitement qui n'est pas spécifique à un langage.

Toutefois, les différents environnements doivent fournir une structure permettant de spécifier les classes contenant les différentes phases programmées par l'utilisateur. Ils

doivent également fournir la possibilité de configurer la phase de l'InputReader et de l'OutputReader, sans compter être capables de transmettre toutes informations qui sont utiles au bon fonctionnement de l'algorithme.

3.2.4 Exécution de la tâche

Au niveau de l'exécution de la tâche, il est important de savoir que l'ensemble des unités de traitement, soit les mappers ou les reducers d'une même phase sont indépendant les uns des autres. Ce fait nous permet alors de pouvoir les exécuter en parallèle, dans des threads différents ou sur des machines distinctes. Cependant, le paradigme MapReduce n'empêche pas également d'effectuer l'exécution d'une tâche de façon séquentielle, c'est-à-dire d'exécuter un à la suite de l'autre les différents mappers et reducers. Pour preuve, l'implémentation de Mr Sim, que nous verrons plus loin, offre le choix d'un flux de travail séquentiel ou parallèle à l'utilisateur. Donc, MapReduce ne se base pas sur une idéologie parallèle, mais il est *propice* au parallélisme.

Jusqu'à maintenant, nous avons toujours parlé d'une tâche n'ayant qu'une seule génération. C'est-à-dire une phase d'InputReader, Mapper, Reducer et d'OutputReader. Toutefois, il est important de savoir que l'on peut créer des tâches multi-générations. Le principe de multi-génération, permet de lancer plusieurs générations une à la suite de l'autre. Les données entrantes utilisées pour la deuxième génération sont les résultats de la première génération, pour la troisième génération les résultats de la deuxième génération, etc. Donc, on peut en conclure que la sortie d'une phase reducer devient l'entrée de la phase

mapper de la génération suivante. De plus, cela implique également qu'il y a une seule phase d'InputReader et d'OutputReader.

3.3 Environnements MapReduce

3.3.1 Mr Sim

Mr. Sim[4] (pour « MapReduce Simulator ») est un environnement de traitement permettant de simuler le fonctionnement d'un environnement de travail MapReduce. Dans les faits, il implémente chaque phase principale du principe MapReduce sous la forme d'un flux de travail (workflow). Le langage d'implémentation est le Java, ce qui permet à Mr. Sim d'être portatif et compatible avec plusieurs outils de programmation et systèmes d'exploitation. De plus, cet environnement a besoin uniquement de l'ensemble de développement Java, JDK, pour être fonctionnel. Il est important de s'apercevoir que cet outil permet donc d'avoir accès à un environnement MapReduce presque directement, comparativement à un environnement Hadoop MapReduce, parce que celui-ci n'a pas besoin d'une installation longue et complexe d'un cluster d'ordinateurs. Mr Sim a été conçu en mettant l'emphase sur l'apprentissage du principe MapReduce et de son utilisation. Au lieu, de ralentir toute recherche dans la création d'un cluster. C'est pour cette raison que Mr Sim est un outil pédagogique intéressant pour toute personne voulant expérimenter le MapReduce.

[4] https://github.com/sylvainhalle/MrSim

Le but premier de cet environnement est d'offrir un outil simple pour créer et tester une tâche MapReduce avec un minimum de configuration. Normalement, l'utilisateur n'a pas besoin d'installer et de configurer des éléments de son ordinateur. Toutefois, sa grande portabilité et sa malléabilité impliquent que l'environnement n'est dans aucun cas optimisé. Il n'est donc pas conseillé de faire des travaux d'exécution sérieux de tâche MapReduce.

D'un point de vue pédagogique, Mr Sim offre des particularités intéressantes, que voici :

- Parce que le traitement est centralisé, il est facile d'effectuer le débogage étape par étape d'une tâche MapReduce (jusqu'à l'implémentation de l'environnement, puisque tout le code source est fourni).
- Il est prêt à être utilisé et à interagir avec un projet de programmation. Il suffit de le placer au bon niveau de l'arborescence de ce dernier pour qu'il soit reconnu.
- L'environnement MapReduce est simulé par environ 300 lignes de codes, ce qui permet à un étudiant qui veut étudier ou s'initier au principe de MapReduce de pouvoir le faire facilement.

Il est important de savoir que Mr Sim, offre des fonctionnalités que les autres environnements MapReduce, tels que Hadoop, n'offrent pas. En voici la liste :

- L'environnement Mr Sim permet d'utiliser le principe d'héritage des objets. Dans Mr Sim, il est possible d'utiliser ce principe lors de la déclaration des types pour les clés et valeurs des tuples. Ceci signifie que le Mapper peut travailler avec des tuples de format (K, V) (où K et V sont des types) peut accepter sans problème un tuple de

format (K', V') si K' est un descendant du type de K et V' est un descendant du type de V. On verra plus loin qu'il est impossible de faire ceci dans un environnement Hadoop, puisque le type des clés et valeurs des tuples doivent être strictement égal aux types du parent.

- Dans Mr Sim, il est possible d'envoyer directement les résultats sortant d'une phase Reducer d'une génération X comme étant les données entrantes d'une phase Mapper de génération X+1. Il est alors possible de sauver du temps d'écriture de la phase de OutputWriter et de relecture de la phase InputReader. L'environnement Hadoop doit en tout temps respecter le fonctionnement du principe MapReduce, même si la tâche nécessite plusieurs générations.

En somme, Mr Sim est un outil intéressant pour faire l'apprentissage du principe MapReduce. De plus, il n'a pas besoin de complexes et coûteux clusters d'ordinateurs pour fonctionner. Sans compter qu'il est un outil libre d'utilisation et est accessible sur sa page Mr Sim.

3.3.1.1 WordCount de Mr. Sim

À titre d'exemple, voici comment le code de l'exemple « word count » décrit précédemment se traduit en code Java utilisant l'environnement MrSim.

```
import java.io.*;
import ca.uqac.dim.mapreduce.*;

public class WordCount
{
  public static void main(String[] args)
  {
    int k = 4; // We keep only words with at least k letters
    int n = 50; // We keep only words that appear n times or more
    SequentialWorkflow<String,String> w =
        new SequentialWorkflow<String,String>( // Initialization
            new CountMap(k), // Mapper
            new CountReduce(n), // Reducer
            new BigStringCollector("data/The-Metamorphosis.txt") // Reader
            );
    // Run the workflow; send results to the InCollector
    InCollector<String,String> results = w.run();
    System.out.println("There are " + results.count() +
        " word(s) of at least " + k +
        " letter(s) that appear at least " + n + " times");
    // Iterator over InCollector to display results
    System.out.println(results);
  }

  private static class BigStringCollector extends Collector<String,String>
  {
    /*package*/ BigStringCollector(String filename)
    {
      super();
      StringBuffer sb = new StringBuffer();
      try
      {
        BufferedReader input =  new BufferedReader(new FileReader(filename));
        try
        {
          String line = null;
          while (( line = input.readLine()) != null)
          {
            sb.append(line.trim()).append(" ");
          }
        }
        finally
        {
          input.close();
        }
      }
      catch (IOException ex)
      {
        ex.printStackTrace();
      }
      super.collect(new Tuple<String,String>(sb.toString(), ""));
    }
  }
```

```
private static class CountMap implements Mapper<String,String>
{
  private int m_minLetters = 1;

  /*package*/ CountMap(int k)
  {
    m_minLetters = k;
  }

  @Override
  public void map(OutCollector<String,String> out, Tuple<String,String> t)
  {
    String[] words = t.getKey().split(" ");
    for (String w : words)
    {
      // Remove punctuation and convert to lowercase
      String new_w = w.toLowerCase();
      new_w = new_w.replaceAll("[^\\w]", "");
      if (new_w.length() >= m_minLetters)
        out.collect(new Tuple<String,String>(new_w, "1"));
    }
  }
}
```

```
private static class CountReduce implements Reducer<String,String>
{
  private int m_numOccurrences = 2;

  /*package*/ CountReduce(int n)
  {
    m_numOccurrences = n;
  }

  @Override
  public void reduce(OutCollector<String,String> out, String key, InCollector<String,String> in)
  {
    int num_words = in.count();
    if (num_words >= m_numOccurrences)
      out.collect(new Tuple<String,String>(key, "" + num_words));
  }
}
```

Figure 5 : Tâche Mr Sim WordCount

3.3.2 Hadoop

Le second environnement MapReduce que nous allons décrire est Hadoop[5], un environnement Java libre qui a été créé par la Apache Foundation dans le but de simplifier la création d'applications distribuées et scalables. Ce dernier permet donc de traiter un grand ensemble de données en utilisant la logique du parallélisme. L'environnement permet d'utiliser des milliers de nœuds et des pétaoctets de données. Ces dernières sont sauvegardées dans des fichiers dans une partition de type HDFS (Hadoop Distributed FileSystem). C'est un système de fichier qui est portable, extensible et qui a été inspiré par le GoogleFS. Le HDFS est écrit en Java et son but ultime est de sauvegarder de très gros volumes de données sur un grand nombre de machines avec des disques durs banalisés. Le système de fichier permet l'abstraction de l'architecture physique de stockage. Cela permet donc de manipuler les données comme il s'agissait d'un seul et unique périphérique de stockage. Il est important de savoir qu'il existe une autre façon de stocker les données dans l'environnement Hadoop. L'application HBase permet de créer une base de données distribuée à travers les nœuds de stockage. Cette dernière est orientée sur le principe de colonnes, c'est-à-dire que l'ensemble des données d'une colonne est stocké sur le même nœud.

L'environnement permet de travailler avec un cluster de nœud de deux types, soit les nœuds maîtres (*master*) et travailleurs (*worker)*. Les nœuds maîtres ont pour tâche de s'occuper de gérer le stockage des données et de coordonner l'exécution des différentes instances de mapper et de reducer. Ce sont eux qui savent où se situent les blocs de données

[5] http://hadoop.apache.org/

à travers des périphériques de stockage. De plus, ce sont eux qui s'occupent de l'exécution de la tâche à travers le cluster. À tout moment, un seul des nœuds maîtres est responsable de l'exécution des tâches. Si jamais celui-ci rencontre un problème, le nœud maître de secours reprend l'exécution de la tâche à l'endroit où l'ancien nœud maître était rendu. Ceci est possible puisqu'il existe un mécanisme permettant de savoir en tout temps où est rendue l'exécution du nœud maître principal pour tous les nœuds de secours. Les nœuds maîtres exécutent le traitement sous le format d'une tâche sous le principe MapReduce. Les nœuds travailleurs eux permettent de stocker les données et effectue le traitement voulu sur ces dernières. Par la suite, quand ils ont terminé leurs tâches, ils attendent le traitement suivant déterminé par le nœud maître.

Sans oublier, qu'il est important de noter que dans l'environnement Hadoop MapReduce, ils nous étaient impossibles d'effectuer un enchaînement de phase Mapper et Reducer dans une même tâche. Donc, pour émuler le multi-générations dans Hadoop, nous avons dû créer un code dynamique permettant de lancer plusieurs tâches une à la suite de l'autre. De plus, cela implique que nous devons avoir une phase d'InputReader et d'OutputWriter pour chaque tâche, ce qui signifie une perte de temps à chaque génération.

3.3.2.1 WordCount de mode Hadoop

À titre d'exemple, voici comment le code de l'exemple « word count », décrit précédemment dans le chapitre, ce traduit en code Java utilisant l'environnement Hadoop MapReduce.

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Figure 6 : Tâche MapReduce WordCount de l'environnement

3.3.3 Mr Sim VS Hadoop

Nous pouvons remarquer qu'il y a quelques éléments qui diffèrent entre les différents outils qui permettent de travailler avec le paradigme MapReduce. Toutefois, nous allons effectuer une comparaison entre Mr Sim et Hadoop pour les points suivants :

- Mr Sim est un outil mono-thread, c'est-à-dire que l'exécution totale de la tâche se fait dans un thread. Donc cela implique alors que l'exécution de la tâche, ainsi que les phases MapReduce, se font une à la suite de l'autre et ce dans l'ordre nécessaire au bon fonctionnement de la tâche. On peut également en conclure que Mr Sim est mono-machine et séquentiel. De plus, un autre fait qui appuie cet énoncé est que Mr Sim est déployable sur une machine, puisqu'il utilise l'outil de développement Java local de la machine. Si on effectue la comparaison directement avec Hadoop, on peut s'apercevoir qu'ils sont totalement différents. La raison est que Hadoop permet l'utilisation d'un environnement distribué. C'est-à-dire de pouvoir utiliser plusieurs unités de traitement qui sont déployées dans un regroupement de machines, c'est-à-dire un cluster, pour exécuter la tâche. De plus, rien n'empêche d'utiliser plusieurs threads par machine pour effectuer un traitement spécifique en parallèle.
- Maintenant au niveau du stockage des données, Mr Sim est totalement dépendant du système de fichier de la machine hôte. De plus, c'est ce dernier qui s'occupe de stocker et de rendre accessible le fichier dans lequel les données sont contenues. Mr Sim ne fait que lire les informations ou en écrire. Ce fait est totalement faux pour

Hadoop, car ce dernier utilise son propre système de fichier, soit le HDFS. De plus, la gestion des fichiers est laissée au nœud maître et c'est ce dernier qui s'occupe de savoir où se trouvent les données dans l'ensemble des partitions HDFS que contient son cluster.

- Mr Sim permet d'utiliser l'héritage lors du transfert de données entre les différentes phases. Pour lui, une instance d'une classe K' qui hérite de K est compatible avec le Mapper ou Reducer pour l'objet K. Ce qui n'est pas le cas dans l'environnement Hadoop. Pour ce dernier, une instance d'une classe K' n'est pas strictement égale à K, donc le Mapper ou Reducer associé à K est invalide pour cette instance. Donc, pour palier au problème, si l'utilisation de plus d'un type d'objet est nécessaire dans les différentes phases, il faut utiliser un objet conteneur qui permet de représenter tous les instances qui héritent de K.
- Le principe d'avoir une tâche multi-générations est possible dans Mr Sim, il suffit de chaîner les Mappers de la génération X aux Reducers de la génération X-1. Toutefois, les derniers Reducers sont liés au OutputWriter pour pouvoir interpréter les résultats du traitement de la tâche. Dans un environnement Hadoop, il est impossible d'effectuer un tel enchaînement. Il faut en tout temps avoir la séquence normale, soit l'InputReader, Mapper, Reducer et OutputWriter, pour effectuer le traitement d'une tâche. Donc, le traitement multi-générations est émulé par le processus principal de la tâche. Celui-ci doit déterminer s'il est nécessaire de lancer une prochaine génération, donner comme fichier d'entrée de la prochaine génération le fichier sortant de la dernière génération et lancer la tâche. C'est

également le processus principal qui doit déterminer quand faire interagir le bon InputReader ou OutputWriter, s'il en existe plus d'une version.

En somme, nous avons décrit les différences majeures entre les deux environnements MapReduce. De plus, c'est sans compter les différences de déclarations des environnements pour créer un objet pour pouvoir lancer la tâche, ainsi que les différences au niveau des ressources disponibles des environnements.

CHAPITRE 4

LTL APPLIQUÉE AU MAPREDUCE

Comme nous avons pu le voir dans la section précédente, il est possible d'utiliser le parallélisme pour décomposer un problème pour qu'il puisse être résolu dans un environnement au moyen du paradigme MapReduce. Toutefois, le problème que nous voulons étudier, soit l'analyse de traces avec la LTL, est resté sans solution jusqu'à tout récemment. En 2009, les chercheurs Kuhtz et Finkbeiner ont démontré que la validation de formules LTL sur des traces appartient à la classe de complexité AC1 (logDCFL) [3]. Leur résultat permet alors d'en déduire que le processus peut être divisé de façon efficace par l'évaluation des blocs entiers d'événements en parallèle. Leur façon de travailler est différente et est accessible au parallélisme, puisque leur technique dépend de la longueur de la trace. De plus, il est important de savoir que leur méthode ne traverse pas les événements de la trace les uns à la suite des autres. Toutefois, lors de l'évaluation du déroulement, ce dernier ne peut être effectué en parallèle. Il faut tout d'abord créer à l'avance un certain type de circuit booléen. Cependant, ce dernier dépend de la longueur de la trace à évaluer.

En outre, la démonstration formelle du résultat montre qu'un nombre déterminé de portes de ce circuit peuvent être sollicités en parallèle à chaque étape du processus, cependant l'algorithme lui-même nécessite un accès global et partagé à la trace entière pour chacun des processus parallèles. Malheureusement, cette idée ne se prête pas directement à un environnement de traitement distribué et parallèle.

Nous allons mettre cette approche de côté et nous allons en étudier une autre. Dans cette section, nous présentons un algorithme qui effectue la validation d'une formule LTL sur une trace en utilisant le principe de MapReduce. L'algorithme effectue le travail d'analyse en itérant la formule LTL sur les données. Lors de la première itération, tous les états qui satisfont les variables propositionnelles, contenu dans la formule à vérifier, sont identifiés. Dans l'itération suivante, les résultats déterminés précédemment sont utilisés pour évaluer toutes les sous-formules qui utilisent directement une de ces variables. Autrement dit, à la fin de l'itération *i* du processus, les événements où toutes les sous-formules de profondeur *i* respectent la propriété sont connus. Il s'ensuit que, pour évaluer une formule LTL de profondeur *n*, l'algorithme aura besoin d'exactement de *n* cycles MapReduce. Chaque cycle MapReduce agit dans les faits comme une forme de testeur temporel [17] en testant une trace constituée d'évaluations de testeurs de niveau inférieur.

Cependant, cela ne signifie pas que la trace doit être lue autant de fois. Dans les faits, la trace de départ est lue uniquement une fois dans son entièreté, lors de la première itération de la procédure. Par la suite, c'est seulement le numéro de référence séquentiel à chaque événements qui doit être transféré entre les mappers et les reducers. Le contenu de la trace originale ne sera plus jamais consulté lors des traitements.

Le système est décrit en fournissant les détails de chacun des composants du principe de MapReduce décrite dans la Figure 1. Nous supposons que chacune des instances du principe (InputReader, Mapper, Reducer et OutputWriter) sont paramétrées par la formule à vérifier φ et la longueur de la trace ℓ.

4.2. Format de la trace et l'InputReader

La phase de l'InputReader est responsable de traiter les parties de la trace et la génération du premier ensemble de tuples clé-valeurs à partir de ces derniers. Nous assumons que chacun des événements est séquentiellement numéroté ou que sa position à l'intérieur de la trace peut être facilement déterminée. Pour un événement *e*, nous allons y référer en utilisant son numéro séquentiel noté #(e). L'algorithme de la phase InputReader, qui est représenté dans la figure 7a, parcourt chaque événement de la partie de la trace qui lui est confiée et évalue sur chaque événement chacune des variables propositionnelles présentes dans φ, la formule LTL à évaluer. Pour chaque variable propositionnelle *a* et chaque événement *e* pour lequel cette variable s'évalue à vrai, l'algorithme émet un tuple <a, (i,0)> ou la variable i représente le numéro séquentiel de l'événement au sein de la trace. On note par atom(φ) l'ensemble de toutes les variables propositionnelles présentes dans φ.

```
Procédure InputReader φ, ℓ (morceau)
    A[]:= atoms(φ)
   Pour chaque e dans morceau faire
       i := #(e)
       Pour chaque a dans A faire
         Si e |= a alors
            émettre <a, (i,0)>
         Fin Si
       Fin
    Fin
```

(a) Pseudo-code pour l'InputReader LTL

```
Procédure Mapper φ,ℓ(<ψ, (n,i)>)
  Si i ≤ δ(ψ)
        S[]:= superformulae(φ,ψ)
    Pour chaque ξ dans S faire
     émettre < ξ , (ψ,n,i + 1)>
    Fin
  Fin Si
```

(b) Pseudo-code pour le Mapper LTL

Figure 7 : InputReader et Mapper

Il est important de remarquer que le traitement initial ne nécessite pas que la trace se situe sur un seul et unique nœud. Chaque nœud contient un fragment de la trace et les événements de ce dernier ne sont pas nécessairement successifs. Il suffit que chacun des événements puisse être remis à sa place séquentielle au sein de l'ensemble de la trace; le nombre de nœuds hôtes qui contient un sous-ensemble de la trace peut être variable. Ceci est particulièrement utile si l'ensemble des événements et leur stockage est fait de façon distribuée.

4.3 Mapper

La phase Mapper a comme format d'entrée des tuples qui ont la forme < ψ,(n,i)>, ces derniers ont pour source la phase InputReader décrite ci-dessus, ou alors les résultats d'un cycle MapReduce précédent. Chacun de ses tuples se lit comme suit : « le processus est à l'itération *i* et la sous-formule ψ est vraie pour l'événement *n* ». On peut voir, en particulier, la façon dont les tuples retournés par l'InputReader expriment ce fait pour les termes de base de la formule à vérifier.

Le pseudo-code du Mapper, que l'on retrouve dans la figure 7b, est responsable de transformer les résultats obtenus pour certaines formules ψ, afin d'obtenir ceux d'une formule ψ' dont ψ est une sous-formule directe (celles-ci sont obtenues en utilisant la fonction superformulæ (φ, ψ)). Par exemple, si les événements où *p* est vraie sont connus, alors ces résultats peuvent être utilisés pour déterminer les événements où **F** *p* est vraie. À cette fin, les Mappers prennent chacun des tuple <ψ,(n,i)> et émettront en sortie un tuple de

la forme <ψ',(ψ,n,i+1)>, où ψ est une sous-formule de ψ'. Ce dernier se lit comme suit : « le processus est à l'itération i + 1, la sous-formule ψ est vrai dans l'événement n et ce résultat est nécessaire pour évaluer ψ' ».

4.4 Reducer

Les mappers sont surtout utilisés pour préparer les résultats de la dernière itération pour être utilisés pour l'itération en cours. En revanche, chaque instance de reducer effectue l'évaluation réelle d'une couche de plus de la formule temporelle linéaire à vérifier. Après l'étape de shuffling, chaque instance individuelle de reducer reçoit tous les tuples de la forme <ψ',(ψ,n,i)> pour une formule ψ', où ψ est une sous-formule directe de ψ'. Par conséquent, le reducer donne toutes les informations sur chacun des numéros d'événements qui respectent ψ' et est en charge de traiter les états où ψ est respecté selon ses informations. Cette tâche peut être décomposée en fonction de l'opérateur principal de ψ'. L'algorithme pour chaque reducer est représenté à la figure 8.

Lorsque la formule de haut niveau à évaluer est **X** ψ, les événements qui satisfont la formule sont ceux qui précèdent immédiatement un événement qui respecte ψ. Donc, le reducer parcourt l'ensemble des tuples d'entrée <**X** ψ, (ψ,n,i)> et produit pour chacun un tuple de sortie <**X** ψ, (n-1,i)>.

Lorsque la formule de haut niveau à évaluer est **F** ψ, les événements qui satisfont la formule sont ceux qui dans le futur respecteront ψ. Le reducer correspondant est celui qui

parcourt les tuples d'entrée et calcule le plus grand numéro d'événement *c* pour lequel ψ est respecté. Tous les événements précédents *c* respectent **F** ψ. Conséquemment, le format que le reducer génère comme tuple de sortie est <**F** ψ,(k,i)>, pour tout k ∈ [0,c].

Le reducer pour ¬ ψ parcourt tous les tuples et garde dans un tableau booléen si $e_i \models \psi$ pour chaque événement *i* de la trace. Par la suite, il émet un tuple <¬ψ, (k,i)>, pour tous les numéros d'événement *k* qui n'ont pas été vu dans les tuples d'entrée. Le reducer pour **G** ψ effectue le traitement dans le sens inverse. Premièrement, il parcourt tous les tuples de la même façon. Si nous définissons *c* comme l'indice du dernier événement pour lequel ψ n'est pas respecté, le reducer émettra des tuples de sortie de format <**G**ψ,(k,i)>, pour k ∈ [c+1, ℓ]. Cela correspond alors à tous les événements pour lesquels **G** ψ est respecté.

Le traitement des opérateurs binaires ∨ et ∧ est un peu plus délicat. Des soins particuliers doivent leurs êtres apportés pour maintenir les tuples dont les résultats seront utilisés dans la dernière itération. Prenons le cas de la formule (**F** p) ∧ q. Les états où les termes clos (feuilles) *p* et *q* sont respectés seront traités par l'InputReader à l'itération 0. Cependant, bien que q soit une sous-formule directe de (**F** p) ∧ q, il faut attendre jusqu'à l'itération 2 pour pouvoir la combiner à **F** p, évaluée à l'itération 1. Plus précisément, un tuple <ψ * ψ', (ψ, n,i)> peut être évaluée uniquement à l'itération δ(ψ * ψ') ; dans tous les itérations précédentes, les tuples <ψ,(n,i)> doivent être remis en circulation. La première condition de l'algorithme des deux reducers prend soin de cette situation. Il est

important de savoir que la fonction delta δ, permet de calculer la profondeur de la propriété demandée. Exemple : La formule δ(G((¬{p0/0}) ∨ (X{p1/0}))) est de profondeur 4.

Dans le cas contraire, lorsque la formule de niveau supérieur à évaluer est ψ ∨ ψ', le format des tuples de sortie est < ψ ∨ ψ', (*n*,i)> chaque fois que le reducer lit des tuples d'entrée < ψ ∨ ψ', (ψ ,*n*,i)> ou < ψ ∨ ψ', (ψ' ,*n*,i)>. Lorsque la formule de niveau supérieure est ψ ∧ ψ', le reducer doit mémoriser les numéros d'événements *n* pour les tuples lus et tels que < ψ ∧ ψ', (ψ ,*n*,i)> et < ψ ∧ ψ', (ψ' ,*n*,i)>, et émettra des tuples de format < ψ ∧ ψ', (*n*,i)> dès qu'il aura vu les deux.

Le dernier cas à considérer est une formule de forme ψ **U** ψ'. Le premier reducer parcourt parmi tous ses tuples d'entrées et mémorise le numéro de l'événement qui respecte ψ et ceux pour lesquels ψ' est respecté. Il procède ensuite à rebours à partir du dernier événement de la trace et émet < ψ **U** ψ', (n,i)> pour un certain état n si ψ' respecte n ou si ψ respecte n et qu'il existe une suite ininterrompue d'états conduisant à un état pour lequel ψ' est respecté. Cette dernière information est traitée via la variable booléenne *b*.

Comme on peut le voir, les tuples produits par chaque reducer ont la forme < ψ, (n,i)>, permettant de porter la même signification que ceux originalement produits par le InputReader, mais pour des formules de plus grande profondeur. Toutefois, le résultat d'un cycle MapReduce peut être réinjecté en entrée pour un nouveau cycle. Comme nous l'avons vue, cela prend exactement δ(φ) cycles complets pour évaluer une formule LTL φ.

```
Procédure Reducer φ,ℓ(F ψ, tuples[])
   m := -1
   Pour chaque <F ψ,(ξ ,n,i)> dans tuples faire
      Si n > m alors m := n
   Fin
   Pour k de 0 à m faire
      émettre <F ψ, (k,i)>
   Fin

Procédure Reducer φ,ℓ (¬ψ, tuples[])
   Pour chaque <¬ψ, (ξ ,n,i)> dans tuples faire
      s[n]:= ⊤
    Fin
    Pour k de 0 à ℓ faire
       Si s[k] ≠ ⊤ alors
          émettre <¬ψ, (k,i)>
       Fin Si
    Fin

Procédure Reducer φ,ℓ (Gψ, tuples[])
   Pour chaque <Gψ, (ξ ,n,i)> dans tuples faire
      s[n]:= ⊤
   Fin
   Pour k de ℓ à 0 faire
     Si s[k] ≠ ⊤ Break
       émettre <Gψ, (k,i)>
     Fin

Procédure Reducer φ,ℓ (ψ ∨ ψ', tuples[])
   Pour chaque <ψ ∨ ψ', (ξ ,n,i)> dans tuples faire
     Si δ (ψ ∨ ψ') ≠ i alors
        émettre < ξ, (n,i)>
     Sinon
        émettre <ψ ∨ ψ', (n,i)>
     Fin Si
   Fin

Procédure Reducer φ,ℓ (X ψ, tuples[])
   Pour chaque <X ψ, (ξ ,n,i)> dans tuples faire
      émettre < ψ ∨ ψ', (n-1,i)>
   Fin

Procédure Reducer φ,ℓ (ψ ∧ ψ', tuples[])
  Pour chaque (ψ ∧ ψ', (ξ ,n,i)> dans tuples faire
   Si δ (ψ ∨ ψ') ≠ i alors
      émettre < ξ, (n,i)>
   Fin Si
   sξ[n]:= ⊤
   Si sψ[n]:= ⊤ et sψ'[n]:= ⊤ alors
      émettre < ψ ∧ ψ', (n,i)>
   Fin

Procédure Reducer φ,ℓ (ψ U ψ', tuples[])
   Pour chaque (ψ U ψ', (ξ ,n,i)> dans tuples faire
     Si δ (ψ U ψ') ≠ i alors
       émettre < ξ, (n,i)>
     Fin Si
     sξ[n]:= ⊤
   Fin
   b:= ⊥
   Pour k de ℓ à 0 faire
      Si sψ'[n]:= ⊤ alors
        émettre <ψ U ψ', (k,i)>
        b:= ⊤
      Sinon Si sψ'[n]:= ⊤ et b = ⊤ alors
        émettre <ψ U ψ', (k,i)>
      Sinon
          b:= ⊥
      Fin Si
     Fin
```

Figure 8 : Pseudo-code pour les reducers LTL

4.5 OutputWriter

À la fin du dernier cycle MapReduce, les résultats sont émis sous le format < φ, (n, δ(φ))>. Ils représentent tous les numéros d'événements *n* tel que $m^n \models \varphi$. Le OutputWriter, représenté dans la figure 9, traduit le dernier ensemble de tuples en la

réponse finale de la formule évaluée. Par la sémantique de LTL, une trace d'événements satisfait une formule φ si $m^0 \models \varphi$. L'OutputWriter indique que la formule est vraie si le tuple < φ, (0, δ(φ))> est trouvé, sinon il écrit qu'elle est fausse.

Procédure OutputWriter φ, ℓ (*tuples*[])
Pour chaque < φ, (n,i)> dans *tuples* **faire**
Si n = 0 **alors**
émettre « La formule est vraie »
Break
Fin Si
Fin
émettre « La formule est fausse »

Figure 9 : Pseudo-code pour le OutputWriter LTL

Pour une trace *m* et une contrainte φ, nous exprimons par $m \models \varphi$ le fait que la trace satisfait la contrainte. Donc, dans le présent contexte, chaque variable propositionnelle est une affirmation de la forme paramètre = valeur, et est évalué à vrai si l'égalité est respectée pour l'événement courant ou à faux le cas échéant.

4.6 Exemple

Nous allons illustrer le fonctionnement de cet algorithme par un exemple simple. Nous considérons la formule suivante :

$$\varphi \equiv \mathbf{G}\ (\neg c \vee \mathbf{F}(a \vee b))$$

évalué sur la trace *a,c,a,d,c,d,b*.

Itération 0

Le InputReader (ou plusieurs InputReaders) traite la trace et génère le premier ensemble de tuples:

<a, (0,0)>,<a, (2,0)>,<b, (6,0)>,<¬c, (0,0)>,<¬c, (2,0)>,<¬c, (3,0)>,<¬c, (5,0)>,<¬c, (6,0)>

Ceci correspond intuitivement au fait que *a* est observée dans les événements de numéro 0 et 2, que *b* est observée à l'événement 6, et que *c* n'est *pas* observée dans les événements 0, 2, 3, 5 et 6.

Itération 1

Les tuples sont alors envoyés aux mappers, qui produisent les tuples suivants:

<a ∨ b, (a,0,1)> , <a ∨ b, (a,2,1)> , <a ∨ b, (b,6,1)> , <¬c ∨ **F** (a ∨ b),(¬c,0,1)> ,

<¬c ∨ **F** (a ∨ b),(¬c,2,1)> , <¬c ∨ **F** (a ∨ b),(¬c,3,1)> , <¬c ∨ **F** (a ∨ b),(¬c,5,1)> ,

<¬c ∨ **F** (a ∨ b),(¬c,6,1)>

Le reducer pour a ∨ b recevra les trois premiers tuples et les tuples émis sont :

<a ∨ b, (0,1)> , <a ∨ b, (2,1)> , <a ∨ b, (6,1)>

On constate qu'effectivement, les événements où soit *a*, soit *b* est observée sont ceux numérotés 0, 2 et 6.

Étant donné que le nombre d'itérations est égal à 1, et que la profondeur de ¬c ∨ **F** (a ∨ b) est de 3, le reducer pour ¬c ∨ **F** (a ∨ b) va tout simplement réémettre ses tuples d'entrée sous le format des tuples suivants :

<¬c, (0,1)> , <¬c, (2,1)> , <¬c, (3,1)> , <¬c, (5,1)> , <¬c, (6,1)≥

On remarque que seul le numéro de l'itération a été incrémenté.

Itération 2

Ces lignes ont été envoyées à des mappers pour un second cycle, les tuples émis sont :

<**F** (a ∨ b),(a ∨ b,0,2)> , <**F** (a ∨ b),(a ∨ b,2,2)> , <**F** (a ∨ b),(a ∨ b,6,2)> ,

<¬c ∨ **F** (a ∨ b),(¬c,0,2)> , <¬c ∨ **F** (a ∨ b),(¬c,2,2)> , <¬c ∨ **F** (a ∨ b),(¬c,3,2)> ,

<¬c ∨ **F** (a ∨ b),(¬c,5,2)> , <¬c ∨ **F** (a ∨ b),(¬c,6,2)>

Le reducer pour la formule **F** (a ∨ b) va émettre :

<**F** (a ∨ b),(0,2)> , <**F** (a ∨ b),(1,2)> , <**F** (a ∨ b),(2,2)> , <**F** (a ∨ b),(3,2)> ,

<**F** (a ∨ b),(4,2)> , <**F** (a ∨ b),(5,2)> , <**F** (a ∨ b),(6,2)>

Tandis que le reducer pour ¬c ∨ **F** (a ∨ b) va de nouveau réémettre :

<¬c,(0,2)> , <¬c,(2,2)> , <¬c,(3,2)> , <¬c,(5,2)> , <¬c,(6,2)>

Itération 3

Les tuples sont envoyés à l'avant-dernier cycle de mappers et ces derniers produiront :

<¬c ∨ **F** (a ∨ b),(**F**(a ∨ b),0,3)> , <¬c ∨ **F** (a ∨ b),(**F**(a ∨ b),1,3)> ,

<¬c ∨ **F** (a ∨ b),(**F**(a ∨ b),2,3)> , <¬c ∨ **F** (a ∨ b),(**F**(a ∨ b),3,3)> ,

<¬c ∨ **F** (a ∨ b),(**F**(a ∨ b),4,3)>, <¬c ∨ **F** (a ∨ b),(**F**(a ∨ b),5,3)> ,

<¬c ∨ **F** (a ∨ b),(**F**(a ∨ b),6,3)> , <¬c ∨ **F** (a ∨ b),(¬c,0,3)> , <¬c ∨ **F** (a ∨ b),(¬c,2,3)> ,

<¬c ∨ **F** (a ∨ b),(¬c,3,3)> , <¬c ∨ **F** (a ∨ b),(¬c,5,3)> , <¬c ∨ **F** (a ∨ b),(¬c,6,3)>

Le reducer pour ¬c ∨ **F** (a ∨ b) produira les résultats suivants :

<¬c ∨ **F** (a ∨ b),(0,3)> , <¬c ∨ **F** (a ∨ b),(1,3)> , <¬c ∨ **F** (a ∨ b),(2,3)> ,

<¬c ∨ **F** (a ∨ b),(3,3)> , <¬c ∨ **F** (a ∨ b),(4,3)> , <¬c ∨ **F** (a ∨ b),(5,3)>,

<¬c ∨ **F** (a ∨ b),(6,3)>

Itération 4

Pour la dernière itération, les mappers produiront les résultats suivants :

<**G** (¬c ∨ **F**(a ∨ b)) , (¬c ∨ **F**(a ∨ b),0,4)> , <**G** (¬c ∨ **F**(a ∨ b)) , (¬c ∨ **F**(a ∨ b),1,4)> ,

<**G** (¬c ∨ **F**(a ∨ b)) , (¬c ∨ **F**(a ∨ b),2,4)> , <**G** (¬c ∨ **F**(a ∨ b)) , (¬c ∨ **F**(a ∨ b),3,4)>,

<**G** (¬c ∨ **F**(a ∨ b)) , (¬c ∨ **F**(a ∨ b),4,4)> , <**G** (¬c ∨ **F**(a ∨ b)) , (¬c ∨ **F**(a ∨ b),5,4)> ,

<**G** (¬c ∨ **F**(a ∨ b)) , (¬c ∨ **F**(a ∨ b),6,4)>

Le reducer pour **G** (¬c ∨ **F**(a ∨ b) émet les tuples suivants suite à son traitement :

<**G** (¬c ∨ **F**(a ∨ b)), (0,4)> , <**G** (¬c ∨ **F**(a ∨ b)), (1,4)> , <**G** (¬c ∨ **F**(a ∨ b)), (2,4)> ,

<**G** (¬c ∨ **F**(a ∨ b)), (3,4)> , <**G** (¬c ∨ **F**(a ∨ b)), (4,4)> , <**G** (¬c ∨ **F**(a ∨ b)), (5,4)> ,

<**G** (¬c ∨ **F**(a ∨ b)), (6,4)>

Finalement, puisque l'événement 0 est un élément de l'ensemble de tuples, l'OutputWriter conclut que la formule est vraie pour la trace analysée. À partir de cet exemple simple, on peut voir comment de multiples InputReader peuvent traiter les différentes parties de la même trace de façon indépendante. Dans les faits, l'InputReader n'a pas besoin que les parties de la trace soient composées d'événements consécutifs au sein de l'ensemble, en autant que chaque événement soit correctement numéroté selon sa position séquentielle au sein de la trace. De plus, un effet secondaire de la mise en place du problème dans le cadre du principe MapReduce, c'est que le nombre d'occurrences de Mapper parallèles qui peuvent être utilisés pour traiter un ensemble de tuples d'entrée à chaque cycle, est variable et non statique. Bref, le nombre d'occurrences varie selon les

besoins de la tâche en cours, sans compter que jusqu'à un Reducer par clé (c'est-à-dire un par sous-formule) peut fonctionner en parallèle dans la phase Reducer d'un cycle.

4.7 LTL-Past

La LTL-Past, la logique temporelle linéaire du passé, est une dérivé de LTL que l'on utilise dans le traitement de l'algorithme en version Hadoop. La différence est que cette dernière prend également en considération les opérateurs de la logique du passé. C'est-à-dire des opérateurs mettant en relation l'événement courant avec des événements antérieurs de la trace. On verra ici comment l'algorithme que nous avons développé peut être étendu à ces opérateurs.

4.7.1 Les ajouts

Pour permettre l'utilisation de la LTL-Past, nous avons ajoutés la logique de ses opérateurs au sein de l'algorithme décrit précédemment. Toutefois, cette ajout est prit en compte uniquement dans la version Hadoop de l'algorithme. Par la suite, nous avons dû ajouter de nouvelles procédures pour les rendre accessibles dans la phase Reducer.

4.7.1.1 Opérateur **S**

L'opérateur **S** signifie « since ». La formule φ **S** ψ est vraie si φ est vrai pour tous les événements précédents jusqu'à ce qu'un événement vérifie ψ. C'est-à-dire que dès que la propriété de gauche a été observée, elle doit l'être pour l'ensemble des événements avant l'événement courant jusqu'à ce que la propriété de droite le soit. L'opérateur **S** est l'inverse

de l'opérateur **U;** ils ont une méthode de traitement semblable, mais un comportement différent à partir de l'événement courant.

4.7.1.2 Opérateur **H**

L'opérateur **H** signifie « historically», ce qui implique par exemple dans le cas de la formule **H** φ que φ est vrai dans tous les événements passés de la trace, à partir de l'événement en cours. C'est-à-dire que la propriété à observer a toujours été vraie dans le passé. L'opérateur **H** est l'opérateur inverse de l'opérateur **G**.

4.7.1.3 Opérateur **O**

L'opérateur **O** signifie « Once ». La formule **O** φ est vraie si φ est vrai dans un événement passé de la trace. Autrement dit, cela veut dire que la propriété sera respectée par un événement passé contenu dans la trace, à partir de l'événement courant. L'opérateur **O** est l'opérateur inverse de l'opérateur **F**.

4.7.1.4 Opérateur **Y**

Finalement, l'opérateur **Y** signifie « yesterday » et la formule est vrai chaque fois que φ est vrai dans l'événement précédent de la trace. Donc, cet opérateur permet d'établir une relation entre l'événement courant et le précédent. L'opérateur **Y** est l'opérateur inverse de l'opérateur **X.**

4.7.2 Les impacts

Suite à l'intégration des opérateurs de la LTL-Past, le plus gros impact est d'augmenter la possibilité de traitement et de rendre possible l'utilisation d'opérateur des deux types de la LTL ensemble. L'algorithme pour chaque reducer est représenté dans la figure 10.

Lorsque la formule de haut niveau à évaluer est **Y** ψ, les événements qui satisfont la formule sont ceux qui succèdent immédiatement un événement qui respecte ψ. Donc, le reducer parcourt l'ensemble des tuples d'entrée <**Y** ψ,(ψ,n,i)> et produit pour chacun un tuple de sortie <**Y** ψ,(n+1,i)>.

Le reducer pour **H** ψ parcourt tous les tuples et garde dans un tableau booléen si e_i ne satisfait pas ψ pour chaque événement i de la trace. Si définissons *c* comme l'indice du dernier événement pour lequel ψ n'est pas respecté, le reducer émettra des tuples de sortie de format <**H** ψ,(k,i)>, pour k ∈ [ℓ, c-1]. Cela correspond alors à tous les événements pour lesquels **H** ψ est respecté.

Lorsque la formule de haut niveau à évaluer est **O** ψ, les événements qui satisfont la formule sont ceux qui dans le passé respectent ψ. Le reducer correspondant est celui qui parcourt les tuples d'entrée et calcule le plus petit numéro d'événement *c* pour lequel ψ est respecté. Tous les événements qui succèdent *c* respectent **O** ψ. Conséquemment, le format que le reducer génère comme tuple de sorties est <**O**ψ,(k,i)>, pour tout k ∈ [c,0].

Le dernier cas à considérer est une formule de forme ψ **S** ψ'. Le premier reducer parcourt tous ses tuples d'entrées et mémorise le numéro de l'événement qui respecte ψ et ceux pour lesquels ψ' est respecté. Il procède ensuite dans l'ordre à partir du premier événement de la trace et émet < ψ **S** ψ', (n,i)> pour un certain événement *n* si ψ' respecte *n* ou si ψ respecte *n* et qu'il existe une suite ininterrompue d'événements conduisant à un événement pour lequel ψ' est respecté. Cette dernière information est conservée via la variable booléenne *b*.

Comme on peut le constater, les tuples produits par les reducers de la LTL-Past ont la même forme que ceux produits par la LTL, soit < ψ, (n,i)>. Ceci leur permet d'être traités dans les mêmes phases et objets de l'algorithme LTLValidator. La LTL-Past permet d'écrire certaines formules plus facilement, cela raccourcit la notation. Il est important de noter que toutes formules écrites avec la LTL-Past peuvent être réécrites avec la LTL. Cependant, pour garder le sens voulu, il faut utiliser un ensemble d'opérateurs et de propriétés pour représenter un opérateur de la LTL-Past.

```
Procédure Reducer φ, ℓ (O ψ, tuples[])
   m := -1
   Pour chaque <O ψ,(ξ ,n,i)> dans tuples faire
      Si n < m alors m := n
   Fin
   Pour k de m à 0 faire
      émettre <O ψ, (k,i)>
   Fin

Procédure Reducer φ,ℓ (Hψ, tuples[])
   Pour chaque <Hψ, (ξ ,n,i)> dans tuples faire
      s[n]:= ⊥
   Fin
   Pour k de 0 à ℓ faire
     Si  s[k] ≠ ⊤  Break
       émettre <Hψ, (k,i)>
     Fin
```

```
Procédure Reducer φ,ℓ (Y ψ, tuples[])
   Pour chaque <Y ψ, (ξ ,n,i)> dans tuples faire
      émettre < ψ ∨ ψ', (n+1,i)>
   Fin

Procédure Reducer φ, ℓ (ψ S ψ', tuples[])
   Pour chaque  (ψ S ψ', (ξ ,n,i)> dans tuples faire
        Si  δ (ψ U ψ') ≠ i alors
   émettre < ξ, (n,i)>
    Fin Si
    sξ[n]:= ⊤
  Fin
  b:= ⊥
  Pour k de 0 à ℓ faire
     Si sψ'[n]:= ⊤   alors
       émettre <ψ S ψ', (k,i)>
     b:= ⊤
     Sinon Si  sψ'[n]:= ⊤ et b = ⊤ alors
         émettre <ψ S ψ', (k,i)>
     Sinon
     b:= ⊥
      Fin Si
   Fin
```

Figure 10 : Pseudo-code pour les reducers LTL-Past

CHAPITRE 5

IMPLÉMENTATION DANS MR. SIM

Dans ce chapitre, nous décrirons comment l'algorithme présenté au chapitre précédent peut être implémenté concrètement en utilisant la première des plateformes MapReduce présentée en début de mémoire, soit MrSim.

5.1 LTLValidator

LTLValidator[6] est un outil et algorithme permettant l'analyse d'une formule LTL sur une trace XML au moyen de MrSim. Son but est de pouvoir déterminer si oui ou non une formule LTL donnée est respectée dans l'ensemble de la trace. Pour ce faire, l'outil utilise comme logique les pseudo-codes préalablement présentés au chapitre 4 pour implémenter les phases du paradigme MapReduce. Pour savoir comment utiliser l'outil LTLValidator, nous référons le lecteur à l'annexe 1 de ce présent document.

5.1.1 Formules LTL

Pour pouvoir utiliser une formule LTL, dans tout système, il faut avoir une structure d'objet permettant de l'écrire et de représenter l'ensemble de ses opérateurs. Dans cette section, nous allons aborder la façon de l'écrire en utilisant les différents niveaux d'objets contenus dans la figure 12.

[6] https://github.com/MaximeSoucy-Boivin/LTLValidator

Pour commencer, nous devons avoir une déclaration de type unique pour l'ensemble des opérateurs. Cela nous permettra d'avoir un seul Mapper et un seul Reducer, chacun acceptant des tuples de ce type général. C'est alors que la classe « Operator » prend tout son sens. Son but est de contenir l'ensemble des fonctionnalités communes à l'ensemble des opérateurs LTL ainsi que la déclaration des fonctions que l'ensemble doit initialiser.

Il est important de noter que la classe « Operator » est abstraite, ce qui veut dire qu'elle ne peut être instanciée. Toutefois, on peut s'apercevoir avec la LTL qu'il existe plus d'un type d'opérateur et que ceux-ci ont des attributs et des méthodes leur étant propres. Donc, pour écrire une formule avec succès, il faut prendre en compte la nature des opérateurs. Pour ce faire, il faut définir, comme le montre le niveau deux de la figure, des classes qui sont des héritiers de la classe abstraite Operator.

Il y a tout d'abord la classe « BinaryOperator » qui représente tous les opérateurs binaires nécessitant un sous-opérateur à gauche et à droite. À l'intérieur de la classe, on retrouve ses deux objets sous les noms de *m_left* et *m_right*, sans compter la chaîne *m_symbol* permettant de contenir la string de l'opérateur ainsi qu'une valeur booléenne permettant de représenter si l'opérateur est commutatif. Cette classe possède également des fonctions lui étant propres, tels que des accesseurs spécialisés pour ses objets de type Operator.

Ensuite, il y a la classe « Atom » qui permet de représenter les variables propositionnelles. Autrement dit, cette classe contient les variables et valeurs à observer

dans la trace. La classe «Atom» possède la chaîne *m_symbol* contenant le nom de l'opérateur, soit dans ce cas-ci la variable « / » suivi de sa valeur, ainsi que des fonctions lui étant propre pour traiter ses données. Il est important de noter que l'on aurait pu implémenter la classe «Atom», sans que celle-ci soit une héritière de la classe abstraite «Operator». Toutefois, cela aurait augmenté la complexité d'écriture des formules LTL. La raison est que les deux autres types d'opérateurs contiennent un ou des opérateurs comme objet et une chaîne de caractère. Alors, il aurait fallu que l'on permette également d'utiliser un objet de type «Atom» et que l'on le traite. Donc, cela aurait eu comme impact d'augmenter la complexité du code et du transfert des données entre les différentes phases de l'algorithme.

La classe « UnaryOperator » permet de représenter tous les opérateurs unaires, c'est-à-dire, qui ne possède qu'une seule sous-formule. Cette dernière est représentée par l'objet *m_operand* et la classe contient également la chaîne *m_symbol* permettant de contenir le nom de l'opérateur.

Les classes que l'on vient de voir nous permettent d'associer la nature des opérateurs et de mieux écrire des formules LTL, mais comment fait-on pour différencier les différents opérateurs ? C'est à ce moment que la nouvelle génération de classe, tel que présenté au niveau trois de la figure, permet de spécialiser le traitement de chaque opérateur. Pour les opérateurs qui héritent de la classe « BinaryOperator » et de la classe « UnaryOperator », la façon de faire ressortir leur caractère unique est pareille. Il faut tout simplement, définir le symbol qui les représente, créer des constructeurs qui permettent de

bien initialiser l'objet qui les représente et de bien définir comment on effectue la comparaison de cet objet. Ensuite, dans le cas du type « Atom », la classe enfant à utiliser est « XpathAtom » et cette dernière contient un tableau de chaîne de caractères : son contenues est le nom de la variable et valeur.

Pour rendre la chose un peu plus claire, voici à la figure 11 un exemple concret de formule avec un arbre des objets utilisés pour la créer :

Exemple 1 : **G**((¬{p0/0}) ∨ (**X**{p1/0}))

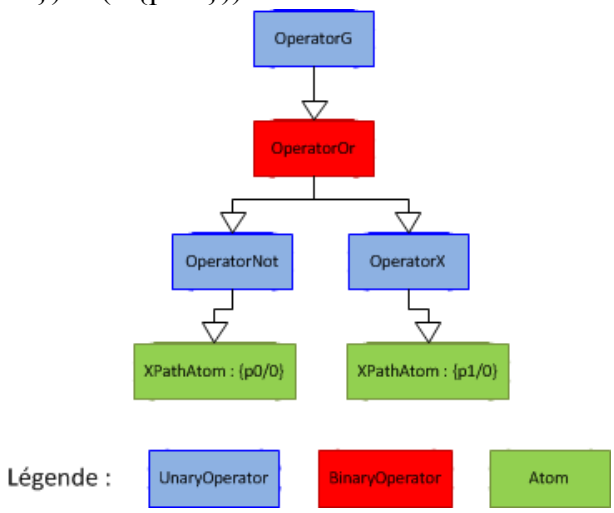


Figure 11 : Arbre de la formule LTL : G((¬{p0/0}) ∨ (X{p1/0}))

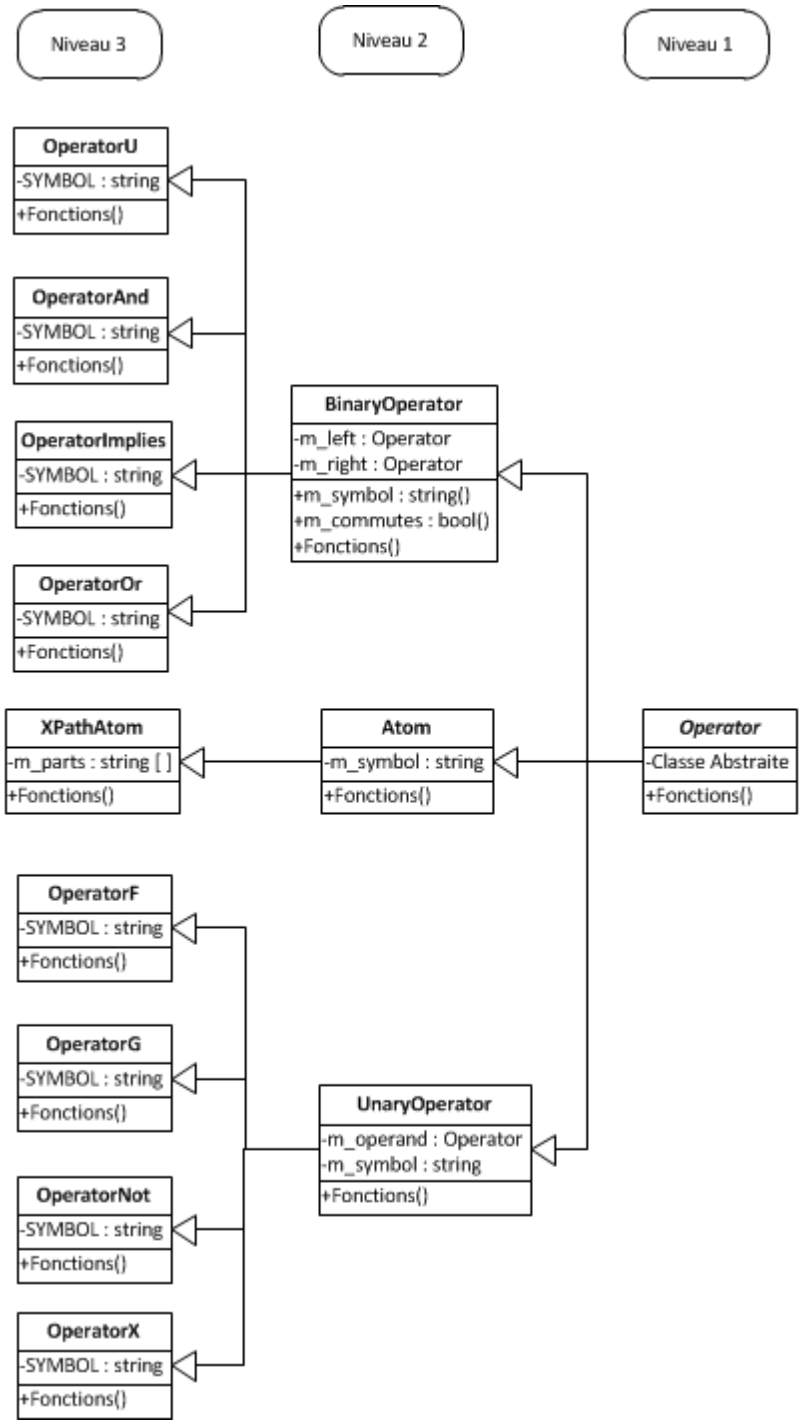


Figure 12 : Diagramme de classes Opérateur LTL

5.1.2 LTLTuple

La classe LTLTuple permet de créer des tuples plus spécifiques au traitement LTL pour contenir les données. La classe Tuple est fournie par Mr Sim et elle permet de contenir une clé de type abstrait *K* et une valeur de type abstrait *V*. Cette classe fournit des accesseurs de valeurs et une méthode permettant d'avoir le tuple dans un format chaîne de caractères (string).

La classe LTLTuple permet d'orienter l'utilisation de Tuple selon la logique LTL. Dans les faits, lors de la déclaration de la classe LTLTuple, la déclaration de la classe Tuple a une clé *K* et une valeur *V* de type bien précis. Dans le cas de la clé, le type spécifié est «Operator», ce qui permettra de pouvoir spécifier un objet opérateur servant à contenir la formule à appliquer sur les données. Pour les données, son type spécifié est de LTLTupleValue, qui à pour but de contenir l'ensemble des données à traiter. Dans ce cas-ci, les données sont l'ensemble des possibilités de contenus que nous avons vus lors de la présentation des pseudo-codes du chapitre 4. Plus précisément : le numéro de l'événement, un opérateur et le numéro de l'itération.

5.1.3 Phase lecture

La phase de lecture des données est assurée par des instances de la classe InputFormat lors de la phase d'InputReader dans des environnements MapReduce distribués. Dans notre cas, la phase est effectuée par des instances de type parser XML qui ont pour tâche de traiter l'ensemble des éléments contenus dans la source de données. Il est important de noter que dans sa version de base de Mr Sim, le seul parser qui était

disponible était de type DOM (Document Object Model). Le type DOM permet de parcourir et manipuler les objets XML sous forme d'arbre. Les objets sont alors tous contenus dans la mémoire vive de la machine. Par la suite, le parser SAX (**S**imple **A**PI for **X**ML) a été ajouté pour pouvoir offrir la possibilité de traiter de plus grand ensemble de données. Le type SAX permet de parcourir les données dans le fichier sans les avoirs dans la mémoire vive de la machine et lorsque nécessaire, il est possibles de parcourir une partie du fichier et non sa totalité.

En premier lieu, le parser DOM lit l'entièreté des éléments à traiter, soit dans notre cas des événements contenus dans un document XML, et les sauvegarde en mémoire. Il est important de savoir que notre parser DOM est dit « intelligent », c'est-à-dire qu'il effectue un traitement en lien avec nos besoins au niveau de la LTL. Le parser émet seulement les paramètres des événements en lien avec la formule à évaluer.

Le fonctionnement du parser SAX est presque identique. La seule différence se situe au niveau de la lecture des données. Le parser SAX lit les événements un par un et les traites immédiatement après les avoir lues. Ce parser a été créé suite à l'apparition du fait que le parser DOM prenait tout l'espace mémoire et que le traitement aboutissait sur une erreur. Cela a donc permis de repousser la possibilité de lecture et de traitement de l'algorithme LTLValidator avec Mr Sim.

5.1.4 Mr Sim parallèle

L'outil Mr Sim a subi une modification importante au niveau des possibilités qu'il offre à l'utilisateur. Il est maintenant possible d'utiliser le parallélisme à l'intérieur de

l'exécution de Mr Sim. Un flux de travail parallèle (ParallelWorkflow) à été ajouté à l'outil, permettant d'effectuer le traitement de l'algorithme en utilisant plus d'un Mapper ou Reducer en même temps. La façon de l'utiliser ressemble beaucoup au flux de travail séquentiel. La seule différence est qu'il utilise des objets de type «ResourceManager» lors de l'instanciation de l'objet.

La classe «ResourceManager» permet de gérer, mais surtout de donner vie au parallélisme dans Mr Sim. Cette classe permet de créer des threads encapsulant le traitement voulu, soit Mapper ou Reducer, et d'ensuite de les retourner. La classe «ParallelWorkflow» reçoit le thread, suite à sa demande faite à l'objet de la classe «ResourceManager», et lance officiellement le traitement du thread. Il est important de noter que la classe « ResourceManager » ne fait pas que créer les threads. Elle gère la disponibilité des threads selon les configurations qu'elle reçoit.

Lors de la création de l'objet, elle reçoit le nombre maximum de threads qu'elle peut créer pour le type auquel elle est associée. Par la suite, cette information lui permet de créer une liste avec comme taille ce dernier. La liste permettra de contenir l'ensemble des threads et de pouvoir les gérer, puisqu'il est important de savoir que l'algorithme peut avoir besoins de plus de threads que le nombre de threads offert. Lorsque tel est le cas, il faut en premier lieu, s'assurer qu'il y a un thread dont son traitement est terminé. Ensuite, cela permettra de le supprimer pour pouvoir créer et ajouter à la liste le nouveau thread demandé par le flux de travail parallèle. Le gestionnaire traite les demandes de threads une à la suite de l'autre et il effectue la création uniquement quand un thread est terminé. Donc, le

gestionnaire restera en mode « trouver un thread terminé », tant qu'il n'en trouvera pas un dont c'est le cas.

En somme, cela aura pour impact de pouvoir comparer directement notre algorithme LTLValidator avec un environnement MapReduce. La raison est que les deux sont compatibles avec le parallélisme. Il suffit donc, pour l'algorithme LTLValidator dans sa version Mr Sim, de revoir la tâche principale et d'utiliser le flux de travail parallèle au lieu de celui séquentiel. Il est important de savoir qu'une version de la tâche en parallèle existe, soit la classe ParaLTLValidation. Finalement, la phase de OutputWriter n'existe pas à proprement dit dans l'outil Mr Sim. Cette phase est émulée par une fonction dans la tâche. Toutefois, son implémentation est basée sur le pseudo-code de l'OutputWriter présenté au chapitre 4.

5.2 Outils connexes

Cette section décrit l'ensemble des différents outils qui sont utiles pour interagir avec l'implémentation de l'algorithme d'analyse. Ces outils permettent de générer des traces à analyser, de faire ressortir les faits des traces et de lancer des tâches d'analyse.

5.2.1 XMLTraceGenerator

XMLTraceGenerator[7] est un outil que nous avons créé pour pouvoir générer un fichier contenant une trace de format XML. Il est important de savoir que l'application génère des événements qui respectent les normes de lecture de l'outil LTLValidator et donc, le format XML décrit au chapitre 2. De plus, il est possible de le paramétrer pour être

[7] https://github.com/MaximeSoucy-Boivin/XMLTraceGenerator

capable de décider l'étendue des noms de variables qui peuvent être contenues dans un événement, puisque les variables ont toujours le format « p » accompagné d'un numéro. De plus, il est possible de paramétrer le type des valeurs, soit numériques ou alphabétiques.

Sinon, une des autres forces de l'outil est de pouvoir créer des fichiers d'une taille voulue. Bref, l'utilisateur détermine à l'avance la taille voulue et l'outil génère le fichier. Toutefois, il est très important de savoir que la génération de deux fichiers de strictement la même taille, contiendra deux traces différentes. La raison est que chaque événement est généré au hasard, puisque sa profondeur est déterminée au hasard entre zéro et le maximum paramétré. Sans oublier que les variables utilisées et leurs valeurs sont déterminées au hasard.

En somme, pour en apprendre plus sur son fonctionnement, veuillez vous référer à l'annexe 2 de ce présent document.

5.2.2 TagCounter

TagCounter[8] est un outil permettant de fournir des faits sur une trace utilisée par LTLValidator. C'est-à-dire qu'il est possible de demander à cet outil combien de fois il retrouve, dans un fichier voulu, un tag XML précis. L'outil émet le nombre de fois qu'il a rencontré le tag à analyser dans l'ensemble du fichier. Pour ce faire, il utilise la logique de lecture Sax avec un « defaulthandlers ».

Finalement, si vous êtes intéressé en apprendre plus sur son fonctionnement, veuillez vous référer à l'annexe 3 de ce présent document.

[8] https://github.com/MaximeSoucy-Boivin/TagCounter

5.2.3 JobLauncher

JobLauncher[9] est un outil permettant de lancer des tâches une à la suite de l'autre de façon automatique. L'outil fonctionne en utilisant des fichiers .bat permettant de spécifier le fonctionnement du jar à exécuter. Plus précisément, dans le fichier bat, se trouve la ligne de commande contenant les paramètres nécessaires qu'un utilisateur devrait utiliser pour lancer la tâche. Il est également important de noter que l'outil possède l'option de pouvoir notifier par courriel l'utilisateur quand l'exécution s'est terminée avec succès. Le courriel contient le temps total d'exécution en millisecondes qui a été nécessaire pour effectuer la tâche ainsi que l'heure de départ et de fin.

Pour en savoir plus sur l'utilisation de cet outil, veuillez vous référer à l'annexe 4 de ce présent document.

[9] https://github.com/MaximeSoucy-Boivin/JobLauncher

CHAPITRE 6

IMPLÉMENTATION DANS HADOOP

Dans ce chapitre nous parlerons des modifications que l'algorithme de traitement et son environnement ont subies pour être compatibles avec Hadoop. Plus spécifiquement, les modifications au niveau des données, du format d'entrée, de transition et de sortie. Nous présenterons également un environnement permettant de programmer et de déboguer l'algorithme dans un environnement Hadoop.

6.1 Environnement de développement

Pour pouvoir programmer une tâche, aussi appelé « Job » MapReduce, il suffit d'avoir un éditeur de texte ou un environnement de programmation qui permet de travailler avec des fichiers JAVA. Toutefois, cela nécessite d'être une personne expérimenté au niveau de la programmation de tâche de format Hadoop MapReduce. Pour tester le code et/ou les réparations des anomalies, l'utilisateur devra recompiler le code chaque fois. Par la suite, il devra le transférer sur le cluster et le copier dans le répertoire nécessaire via des lignes de commandes, ce qui peut être ardu comme opération pour un programmeur. Cette façon de faire fournit très peu d'informations sur le pourquoi la tâche n'effectue pas le traitement comme voulu : le programmeur ne dispose que des journaux de l'environnement pour le savoir. Toutefois, ces derniers sont créés seulement pour dire à partir de quand la

tâche génère une erreur et avec quelle information. Donc, si l'erreur se situe au niveau de la logique, il sera difficile de la détecter via les journaux.

Il est donc conseillé d'utiliser un environnement de programmation permettant de pouvoir exécuter le code sans avoir besoin d'un cluster d'exécution. Pour ce faire, il suffit d'avoir un ordinateur ou machine virtuelle utilisant comme système d'exploitation Ubuntu 12.04 LTS et un environnement de programmation Eclipse. Pour savoir comment effectuer la configuration et utiliser un tel environnement, on se réfèrera à l'annexe 5 de ce mémoire.

6.2 Structure XML

Avec l'outil LTLValidator, un enregistrement est un événement complet même si ce dernier était sur plusieurs lignes. Dans le cas de Hadoop, cette hypothèse n'est plus valide. Mais qu'est-ce qu'un enregistrement pour Hadoop ? Il s'agit d'éléments contenus entre deux caractères de fin de ligne. Ceci implique qu'une ligne est un enregistrement, et qu'un même événement pourrait voir l'ensemble de son contenu séparé dans des parties différentes de la trace, qui se retrouveront dans deux nœuds de traitement différents.

Pour une utilisation dans un environnement Hadoop MapReduce, le format XML utilisé jusqu'ici dans ce mémoire n'est pas conseillé. La raison est que celui-ci n'a pas été du tout pensé pour cet environnement et cet usage. Donc, il faut lui apporter des modifications pour le rendre viable pour pouvoir ainsi utiliser les données. Le nouveau format, Hadoop MapReduce est présenté dans la figure 13b. Pour simplifier le traitement, il a fallu trouver une solution permettant de rendre accessible dans l'ensemble des nœuds de traitements la taille totale de la trace. Donc, la balise TraceLenght a été créée pour contenir

le total d'événements de la trace. Toutefois, il est important de savoir que, par défaut, l'ensemble des nœuds n'ont pas nécessairement cette information. Alors, le nœud maître s'occupe de récupérer l'information et de la retransmettre à l'ensemble des nœuds via un paramètre de l'objet de la tâche.

Ensuite, nous avons dû ajouter une balise dénommée TL pour permettre de bien identifier le numéro de l'événement au sein de la trace. La raison est qu'il faut bien se rappeler que la trace est divisée en parties. Donc, l'ordre propre des événements au sein de la trace ne sera pas respecté. Sinon, le dernier aspect et le plus important est le format ligne. L'environnement MapReduce sépare la trace en parties de longueur aussi égales que possible. Il divise la trace à la fin d'un enregistrement et non au milieu.

Pour générer une trace en format Hadoop, il existe un dérivé de l'outil XMLTraceGenenrator, soit HadoopTraceGenerator[10], qui implémente les différences précédemment décrites. La façon de les utiliser est identique et donc, si vous voulez l'utilisez veuillez vous référez à l'annexe 2.

[10] https://github.com/MaximeSoucy-Boivin/HadoopTraceGenerator

```
<Trace>

<Event>
<p2>21</p2>
<p0>0</p0>
</Event>

<Event>
<p4>6</p4>
<p2>21</p2>
<p0>0</p0>
</Event>

<Event>
<p4>2</p4>
<p0>0</p0>
</Event>

</Trace>
```

```
<Trace>

<TraceLength>6</TraceLength>

<Event><TL>0</TL><p2>21</p2>,<p0>0</p0></Event>

<Event><TL>1</TL><p4>6</p4>,<p2>21</p2>,<p0>0</p0></Event>

<Event><TL>2</TL><p4>2</p4>,<p0>0</p0></Event>

<Event><TL>3</TL><p0>0</p0></Event>

<Event><TL>4</TL><p5>6</p5>,<p7>2</p7>,<p0>0</p0></Event>

<Event><TL>5</TL><p0>0</p0></Event>

</Trace>
```

(a) Traditionnel VS (b) Hadoop MapReduce

Figure 13 : Trace XML traditionnelle

6.3 Format d'entrée de Hadoop

Outre la structure des traces XML elle-mêmes, d'autres modifications ont dû être apportées pour que l'environnement Hadoop puisse lire leurs contenus et le résultat de phases MapReduce subséquentes.

6.3.1 FileInputFormat

Le FileInputFormat est la classe de base pour tous les formats d'entrée dans un environnement Hadoop MapReduce. Pour qu'une classe soit considérée comme étant un format d'entrée, elle doit avoir hérité de cette dernière et avoir d'implémenter les fonctions

de base du traitement. Il existe plusieurs formats de base fournis avec l'environnement et prêts à être utilisés. Voici les types les plus utilisés.

TextInputFormat <K, V> est le format d'entrée par défaut. Les enregistrements produits contiennent une clé *K* de type *LongWritable*, qui représente le nombre d'octets à partir du début du fichier. La valeur *V* de type *Text*, contient la valeur contenue dans l'enregistrement, à l'exclusion du saut de ligne. Pour mieux comprendre le fonctionnement, voici un exemple tiré du livre Hadoop The definitive Guide :

> On the top of the Crumpetty Tree
> The Quangle Wangle sat,
> But his face you could not see,
> On account of his Beaver Hat.

Cet exemple sera divisé en 4 enregistrements. Ces derniers seront interprétés dans le format clé-valeur comme suit :

> (0, On the top of the Crumpetty Tree)
> (33, The Quangle Wangle sat,)
> (57, But his face you could not see,)
> (89, On account of his Beaver Hat.)

SequenceFileInputFormat est un format d'entrée permettant à Hadoop d'utiliser les fichiers binaires. Les éléments contenus dans un fichier sont des séquences de paires clé-valeur binaires. C'est un format qui est bien adapté à un environnement Hadoop MapReduce, puisqu'ils sont divisibles facilement. La raison est qu'ils ont des points de synchronisation afin que les lecteurs puissent synchroniser avec les limites des enregistrements à partir d'un point arbitraire dans le fichier, comme le début d'une partie.

De plus, c'est un format qui prend en charge la compression et il peut stocker des types de données arbitraires en utilisant une variété de cadres de sérialisation.

Les autres types les plus populaires sont *FixedLengthInputFormat*, *KeyValueTextInputFormat*, *MultiFileInputFormat* et *NLineInputFormat*. Ils sont créés pour effectuer des tâches plus pointues au niveau du traitement des données.

6.3.2 LTLInputFormat1Gen

LTLInputFormat1Gen est un format d'entrée que nous avons dû créer pour répondre à nos besoins de traitement pour la première génération de notre traitement multi-générations. La raison est qu'il n'existe aucun format capable de traiter le format de nos données et de les mettre dans des tuples appropriés pour nos traitements. Les éléments retournés sont un LongWritable et une liste contenant uniquement des TupleHadoop<Atom, LTLTupleValue>. La variable LongWritable est la clé du couple de valeur émis et sert uniquement à identifier de façon unique chaque événement. Il est important de noter que ce numéro est en réalité le numéro de l'événement contenu dans le tag TL. Sinon, les objets de TupleHadoop permettent de contenir la sous-formule et la taille totale de la trace. De plus, il est important de savoir que c'est un objet qui hérite de la classe Tuple. Ce dernier sert à garder les informations utiles à l'analyse, c'est-à-dire l'élément en lien avec les données, soit un *Atom*, et les valeurs du tuple, soit les objets *LTLTupleValue*.

Il est important de savoir que c'est un format d'entrée Hadoop « intelligent ». C'est-à-dire que le format détermine si oui ou non, un événement doit demeurer pour aller à la prochaine étape. Pour déterminer ce fait, il utilise les atomes contenus dans la formule et compare chaque variable de chaque événement à chacun des atomes. Si un événement contient une ou plusieurs variables qui est contenue dans les atomes, alors chacune des variables sera transformée en un objet TupleHadoop et sera placée dans la liste correspondante.

6.3.3 LTLInputFormatXGen

LTLInputFormatXGen est un format d'entrée qui a pour but de traiter les données d'entrée de la deuxième à la dernière génération. Son fonctionnement est différent de la phase de lecture précédente, puisqu'elle part des résultats de la génération précédente. Il est important de savoir que le format n'a pas besoin d'effectuer de traitement «intelligent», puisque l'ensemble des éléments contenus dans le fichier doit être traité par l'algorithme lors de la génération courante. Ce type de format retourne des objets de type *OperatorContainer*, servant à contenir l'opérateur qui représentera la clé du tuple, ainsi que les valeurs qui seront contenues dans un objet de type TupleHadoop<Operator, LTLTupleValue>. Ce dernier objet contient également un ensemble contenant les sous-formules à étudier ainsi que la taille totale de la trace.

De plus, il est également important de savoir que le format d'entrée détecte qu'il n'y a aucune information à traiter pour cette génération, via une phrase clé programmée dans la

phase d'OutputWriter. Si tel est le cas, la phase de lecture est arrêtée et le traitement associé à cette génération se termine.

6.4 Format des objets de transitions

Le format des objets de transitions est tout simplement le format que doivent avoir les objets transmis entre les phases Mapper et Reducer. Il est important de noter qu'entre les autres phases ce format n'est pas obligatoire, mais est toutefois conseillé. Ce format permet de sérialiser les objets pour faciliter le transfert réseau. Il existe deux types d'objets soit le type Writable et WritableComparable.

6.4.1 Writable

Le type Writable permet de définir un objet de base transférable entre des phases MapReduce de l'environnement Hadoop. Il est dit de base, puisque ce dernier ne fait que rendre l'objet sérialisable. Lorsqu'un objet hérite de ce dernier, celui-ci doit implémenter les méthodes de la classe Writable.

Ainsi, la méthode « write » permet, lorsque l'objet est appelé à être transféré sur le réseau, d'écrire une à la suite de l'autre les informations contenues dans l'objet. L'écriture des informations est faite en octets, ce qui permet d'optimiser le transfert des données. Les informations restent dans cet état jusqu'à ce que la phase suivante reçoive les informations et les interprète. C'est alors que la fonction « readFields » entre en jeu. Son but est de prendre les informations en format octets et de les convertir dans leurs formats d'origine. Toutefois, il est important de noter que l'ordre dans lequel les informations ont été écrites doit être également l'ordre dans lequel les informations sont réinterprétées.

Ce fonctionnement a été utilisé pour la création de l'objet TupleContainer de notre algorithme. Son but est de contenir et gérer tous les tuples possibles, tout en permettant de pouvoir les sérialiser sans perdre d'information.

6.4.2 WritableComparable

Le type WritableComparable est majoritairement semblable à l'autre type. C'est-à-dire qu'il doit aussi être utilisé comme parent d'une classe, doit avoir les méthodes «readFields» et «write» comme décrit plus haut . Toutefois, la différence ce situe au niveau du fait que ce type est conçu pour être, comme son nom l'indique comparable. La classe héritière contient la fonction « compareTo », qui implémente la façon dont doit se faire la comparaison.

Le but de ce type est pourvoir créer des clés sérialisables et classables pour l'environnement Hadoop MapReduce. Il est important de rappeler qu'entre les phases de Mapper et Reducer, il y a la sous-phase de Shuffling, qui a pour tâche de regrouper les tuples en fonction de leur clé ; pour ce faire, l'environnement Hadoop utilise la fonction « compareTo ». Un fonctionnement de ce type a été utilisé lors de la création de l'objet OperatorContainer.

6.5 Format de sortie

Le format de sortie est un type de classe permettant de spécifier comment écrire les données de fin de traitement. Pour ce faire, il faut créer une classe ayant comme parent le type FileOutputFormat de l'environnement Hadoop. Ce dernier type permet de fournir un

standard de classe pour écrire des objets, de manière à ce que l'environnement puisse les utiliser lors de la phase d'OutputWriter.

6.5.1 LTLOutputFormatXGen

La classe LTLOutputFormatXGen permet d'écrire les résultats de traitement de la première à l'avant-dernière génération de notre algorithme. Pour ce faire, elle utilise les méthodes de base de la classe parent. Elle implémente la méthode « write », « close » et «getRecordWriter» . La méthode « write » permet d'écrire chacun des tuples clé-valeurs avec l'ensemble de leurs données. Pour ce faire, elle s'occupe d'établir la structure des résultats, soit comme balise ouvrante et fermante le contenu de l'opérateur clé. Les données sont inscrites entre les deux balises, puisque la balise fermante est écrite en dernier. Il est important de noter que toutes les informations sont contenues sur la même ligne. Les données écrites sont déterminées via le type de l'objet qui contient les données. Donc, selon sa structure, les différentes données sont inscrites.

La méthode « close » permet d'écrire les informations de fin de traitement et de fermer le flux d'écriture sortant. Il est important de noter que le contenu de cette méthode s'adapte au besoin de traitement de la tâche. Donc, il peut écrire tous les messages dont toutes tâches ont besoin. Sinon, la méthode « getRecordWriter » permet d'initialiser et d'acquérir un objet de cette classe pour pouvoir effectuer le traitement voulu. Cette classe est donc utilisable et accessible par la phase d'OutputWriter pour l'environnement Hadoop.

6.5.2 LTLOutputFormatLastGen

La classe LTLOutputFormatLastGen est presque identique au type précédemment décrit. La différence majeure se situe de la méthode « write », puisque cette dernière effectue le test pour savoir si la formule est vraie sur le tuple reçu. Par la suite, elle effectue l'écriture des informations envoyées. Il est important de noter que dès que la formule est évaluée à vrai elle le sera peut importe s'il y a d'autres tuple à traiter ou non. La seconde et dernière différence se situe au niveau de la méthode « close », puisqu'elle doit prendre en considération si la formule est vraie ou fausse pour écrire les informations de fin de traitement. Finalement, ce format de sortie est utilisé seulement au niveau de la dernière génération de la tâche comme étant le format de sortie des données de fin de traitement.

6.6 Instanciation de la tâche

Pour pouvoir lancer une tâche MapReduce, il faut tout d'abord instancier un objet pour la représenter. Le but de ce dernier est de bien sûr, déterminer la source de données, la classe Mapper et Reducer à utiliser, sans oublier la classe d'InputReader et d'OutputWriter. Il est important de noter ce que l'on entend par tâche est une séquence de phases. Donc, si la tâche complète exige plus d'une séquence, alors il faudra créer plusieurs objets pour pouvoir la représenter et les inters reliés ensemble.

6.6.1 L'objet jobConf

L'objet jobConf permet, dans un environnement Hadoop, de créer une entité permettant de configurer une tâche et de pouvoir la lancer au moment voulu. Cet objet est celui que l'environnement rend disponible à l'utilisateur pour lancer une tâche et interagir

avec l'environnement. Donc, l'utilisateur doit absolument utiliser cette ressource, puisqu'il est impossible de déclarer un autre objet comme étant descendant de celui-ci.

Pour instancier l'objet avec succès (voir un exemple à la figure 14), on doit tout d'abord créer un nouvel objet avec son constructeur et lui spécifier la classe représentant la tâche à effectuer. Par la suite, il faut donner un nom unique à la tâche via l'option «setJobName».

Ensuite, il faut définir le format d'entrée de Hadoop pour que la phase de lecture des données s'exécute et soit compatible avec la phase Mapper. Pour ce faire, il faut utiliser l'option « setInputFormat ». De plus, il est possible de fournir des informations à la tâche via des paramètres grâce aux options « set », « setStrings » et « setInt ». Il nous reste seulement pour la phase Mapper, de configurer la classe à utiliser via « setMapperClass ».

Par la suite, il faut spécifier le fichier contenant les données d'entrée via l'option «setInputPaths». Il faut également spécifier le fichier que l'environnement utilisera pour écrire les informations de fin de traitement lors de la phase d'OutputWriter. Pour ce faire, l'option à utiliser est « setOutputPath ».

Finalement, il nous reste à spécifier la classe du Reducer à l'aide de l'option « setReducerClass » et de spécifier le type de sa clé par « setOutputKeyClass » et le type des données par « setOutputValueClass ». De plus, il ne faut pas oublier de spécifier le format de sortie via « setOutputFormat ».

```
//Start of init for the mapreduce job
JobConf conf = new JobConf(LTLValidation.class);
conf.setJobName("LTLValidation Generation :" + i);
conf.setStrings("property_str", property_str);

conf.setMapperClass(LTLMapper1Gen.class);
conf.setInputFormat(LTLInputFormat1Gen.class);
conf.set("xmlinput.start", "<Event>");
conf.set("xmlinput.end", "</Event>");
conf.set("tracleLength.start", "<TL>");
conf.set("tracleLength.end", "</TL>");
conf.set("baseName.start", "<p");
conf.set("baseName.end", "</p");
conf.setInt("m_traceLength",traceLength);

FileInputFormat.setInputPaths(conf, new Path(trace_filename));
outPath = new Path(OutputFolder + "/Gen" + "0" + i);
FileOutputFormat.setOutputPath(conf, outPath);

//Output for the reducer
conf.setReducerClass(LTLReducer.class);
conf.setOutputKeyClass(OperatorContainer.class);
conf.setOutputValueClass(TupleContainer.class);
conf.setOutputFormat(LTLOutputFormatXGen.class);
```

Figure 14 : Exemple de l'objet

6.6.2 Les formats de sortie

Au niveau des formats de sorties, il est important de savoir qu'il faut en tout temps que le type de sortie du Mapper soit le même type de celui d'entrée du Reducer. Sinon, l'environnement génère un message d'erreur spécifiant qu'il n'a pas de Reducer correspondant. Cette règle est obligatoire entre les phases, mais les types sont strictement comparés entre la phase Mapper et Reducer. Sinon, entre les autres phases, il est possible de spécifier des types d'objets génériques qui permettent d'accepter tous les objets.

Sinon, il est également important de respecter les formats configurés dans l'objet de la tâche. Si ce n'est pas le cas, l'environnement générera un message d'erreur indiquant que le type est invalide. Donc, il est souhaitable de toujours de déclarer le prototype de la phase suivante avant de finir la programmation de la phase en cours.

CHAPITRE 7

CLUSTER HADOOP MAPREDUCE

Dans ce chapitre, nous allons voir en détail deux différentes structures permettant de créer un cluster Hadoop MapReduce de référence, c'est-à-dire un nœud Maître avec trois nœuds Travailleurs. Dans chacune des sous-sections, nous présentons les différentes applications ou outils utilisés pour créer le cluster.

7.2 L'univers VMware

La compagnie VMware propose une multitude de produits propriétaires permettant la virtualisation d'architectures. C'est-à-dire des applications permettant d'accueillir différents systèmes d'exploitation ou applications spécialisées, tout en permettant de partager les ressources d'une machine physique entre les différentes machines virtuelles.

7.2.1 Project Serengeti

Le Project Serengeti est un projet open-source permettant le déploiement rapide d'un environnement Hadoop sur une plate-forme virtuelle VMware. Il a été initié par VMware et est strictement compatible avec un environnement VSphere. Project Serengeti permet de déployer le système de fichier de Hadoop, HDFS (Hadoop Distributed FileSystem), sur l'ensemble des nœuds du cluster de façon simple et rapide. De plus,

Serengeti est complètement compatible avec le modèle de programmation MapReduce. Il est de plus possible d'utiliser Pig, une plate-forme qui est composée d'un langage de haut niveau et qui permet l'analyse de grands ensembles de données. Ensuite, il est possible d'utiliser le logiciel d'entrepôt de données Hive, qui permet d'exécuter des requêtes SQL sur les entrepôts de données HDFS. Le projet permet également l'utilisation de HBase, qui est une base de données orientée colonne et distribuée construite sur le HDFS. Project Serengeti est approuvé par les principaux distributeurs de Hadoop, soit, Cloudera, Hortonworks, Intel, MapR, et Pivotal.

Project Serengeti est disponible en plusieurs versions, 0.5-0.6-0.7-0.8, sur le site de VMWare. Il est important de noter que depuis septembre 2013, le projet a évolué et est devenu Big Data Extension 1.0. Cette version est la dernière mouture de Project Serengeti, avec moins de bogues et davantage de stabilité. Il est important de savoir que cette version supporte encore mieux les fonctionnalités natives de VSphere.

7.2.2 VMware VSphere

VSphere est une plate-forme de virtualisation permettant la création d'infrastructures du Cloud. Autrement dit, c'est un hyperviseur de type 1, « *bare metal* », basé sur l'architecture VMware ESXi. Il est important de noter qu'un hyperviseur est une plate-forme permettant à plusieurs systèmes d'exploitation de s'exécuter en même temps, tout en se partageant les différentes ressources disponibles. De plus, un hyperviseur de type 1 est dit également natif, puisque ce dernier est une application qui est directement déployée sur la plate-forme matérielle de l'hôte. L'hyperviseur est, autrement dit, un noyau

spécialisé pour faire exécuter des systèmes d'exploitation invités sur l'architecture pour lequel ils sont compatibles et optimisés.

7.2.3 VMware VCenter Server

VCenter est une application permettant de gérer l'ensemble des serveurs ainsi que l'univers virtuel des entreprises. Ce dernier offre une plate-forme regroupant un ensemble de fonctionnalités et d'outils fort utiles, tels le déploiement simplifié, l'utilisation de profils d'hôtes, l'authentification centralisée, la personnification d'autorisations ou de rôles, la gestion des ressources, les redémarrages automatiques, etc. De plus, VCenter vient avec des sous-applications qui ont pour but d'aider l'utilisateur avec sa structure de serveurs. Parmi celles-ci notons VCenter Orchestrator, qui permet d'automatiser plus de 800 tâches grâce aux workflows, le tout dans une interface graphique. Ensuite, il y a VCenter Multi-Hypervisor Manager qui permet de simplifier la gestion et le contrôle d'un regroupement d'hyperviseurs. Finalement, il y a le VCenter Operations Management Suite, qui a pour tâche d'améliorer la santé et les performances générales de l'infrastructure VSphere. Pour réussir cela, il dispose d'une grande visibilité sur les opérations qui s'effectuent dans l'infrastructure.

7.2.4 Windows Server 2008

Ce dernier est un système d'exploitation orienté serveur. Il permet alors, selon le principe de serveur, d'introduire l'idéologie et l'utilisation de *services,* ce qui permet alors de pouvoir exécuter différentes applications séparément, ensemble ou en même temps. De plus, le système d'exploitation est conçu pour aider les gestionnaires à gérer les problèmes

via des outils de diagnostic, sans oublier qu'il permet de gérer un grand trafic réseau entrant et sortant. Il est important de noter que le but premier d'un tel Windows est l'utilisation des ressources logicielles et matérielles de la machine par d'autres machines. Finalement, il est bon de savoir que le système d'exploitation est compatible avec la majorité des applications de type Windows.

7.2.5 Le déploiement du cluster

Après de nombreux essais et erreurs, nous en sommes venus à la conclusion qu'il ne faut pas uniquement déployer Project Serengeti dans des machines virtuelles VMware pour pouvoir l'utiliser. Dans les faits, il faut installer de façon native un système d'exploitation de type Serveur sur la machine de test. Nous avons décidé d'utiliser le système d'exploitation Windows Server 2008. Ensuite, il a fallu installer l'application de gestions des entités VMware, soit VCenter sur notre système d'exploitation serveur. VCenter nous permet de gérer et d'utiliser l'entité VSphere dont nous avons besoin pour déployer Serengeti. Toutefois, il faut installer cette entité dans une machine virtuelle, puisque nous n'avons pas d'autres ordinateurs disponibles et qu'il est impossible de faire exécuter deux systèmes d'exploitation en même temps de façon native sur une même machine.

Il est alors possible d'effectuer un déploiement de Serengeti. Toutefois, il est important de noter que la façon de déployer Seregeti dans VSphere via VCenter est facile. Cependant, il nous avons été dans l'impossibilité de déployer un cluster fonctionnel en utilisant Serengeti. Le premier problème rencontré est que les images à utiliser pour déployer un cluster nous demandent un nom d'utilisateur et mot de passe pour se connecter.

Cette information ne nous a pas été transférée lors du téléchargement de Serengeti; il nous a fallu faire des recherches via Google pour la trouver. Par la suite, les principaux problèmes que nous avons rencontrés sont que les machines virtuelles déployées à partir des images de Serengeti avaient un problème réseau majeur ou qu'une d'entre elles était corrompue. Malheureusement, le nœud Maître était souvent atteint de l'un de ses problèmes. Donc, la création d'un cluster via Serengeti a dû être abandonnée malgré le grand nombre de tentatives, faisant varier à chaque fois le nombre d'entités VSphere ou différents paramètres de configuration.

Finalement, nous en sommes venus à la conclusion que notre problème était dû au minimalisme des ressources matérielles, puisque pour contenir notre entité VSphere, nous avons été dans l'obligation d'utiliser une machine VMware dans un disque dur externe. De plus, l'entité VSphere était lancée sur la même machine que VCenter, ce qui diminuait les ressources disponibles. Ensuite, il est important de noter que VMware se base sur les adresses IP pour effectuer le transfert de données, ce qui produit des tâches inutiles pour l'ordinateur : ce dernier doit transférer les données à un nœud qui est contenu en lui-même. Nos recherches nous ont permis de constater que les personnes qui utilisent Serengeti possèdent en général de « vrais » serveurs avec des disques SSD pour créer leur cluster. Or dans notre cas, la machine de recherche est un ordinateur possédant de grandes capacités de traitements, comparativement à la moyenne des ordinateurs présentement disponibles.

7.3 L'univers Solaris

Une autre avenue s'est offerte à nous pour créer un cluster Hadoop. C'est à ce moment que nous avons découvert la possibilité de créer un cluster Oracle Solaris 11.1. Nous allons voir dans les prochaines sous-sections les éléments à utiliser pour le créer et parler des avantages de cette façon de faire.

7.3.1 Oracle Solaris 11.1

Le système d'exploitation Oracle Solaris 11.1 est un système d'exploitation de la famille UNIX. Il est reconnu pour son évolutivité, en particulier sur les systèmes SPARC. Il intègre de nouvelles fonctionnalités innovantes telles que l'outil Dtrace qui permet de déboguer en temps réel des problèmes d'applications et son système de gestion de fichiers ZFS qui permet de mieux gérer et d'optimiser le stockage des données. De plus, ce système d'exploitation prend en charge les postes de travail x86, serveurs SPARC, serveurs d'Oracle et d'autres fournisseurs.

C'est un système d'exploitation propriétaires développé par la compagnie Oracle. Il a été écrit en utilisant les langages de programmation C/C++ et est un système d'exploitation comprenant un noyau monolithique avec des modules alloués dynamiquement. Il est possible de l'utiliser sans avoir besoin d'une licence provenant d'Oracle. Lors de l'installation, le processus ne demande aucun numéro de série ou de licence pour l'activer et terminer l'installation. S'il est un système ouvert au niveau de l'utilisation, il est cependant impossible d'accéder au code de ce dernier.

7.3.2 Oracle VM VirtualBox

Oracle VM VirtualBox est une application permettant de créer des machines virtuelles. Il est important de noter que l'application est compatible avec les systèmes d'exploitation Windows, Unix, Linux, OS X d'Apple et Oracle. L'application est compatible avec les processeurs de type AMD64 et Intel64. De plus, elle est une application pour usage autant industriel que personnel.

Cela nous permettra de créer un cluster Hadoop MapReduce à l'aide du système d'exploitation Oracle Solaris 11.1. Un de ses avantages est qu'il ne perdra pas sa connexion Internet. Il a en effet été observé, lorsque le système d'exploitation est installé de façon native, que le système d'exploitation perd le type de sa connexion et n'est plus capable de se reconnecter. Pour régler le problème, il aurait fallu avoir accès à la configuration du réseau en tant qu'administrateur.

7.3.3 Observation sur le tutoriel

Pour créer un cluster Hadoop MapReduce, il existe un tutoriel[11] qui fournit l'ensemble des étapes à exécuter. Tout d'abord, il explique comment installer l'application VirtualBox. Par la suite, il donne étape par étape les lignes de commandes à faire. De plus, il fournit des explications détaillées lorsque nécessaire. Dans ce tutoriel, la version installée de Hadoop MapReduce est 1.2.1. Il est important de noter qu'il est possible d'utiliser une autre version de Hadoop, toutefois il n'est pas garanti que le tutoriel sera complet. Dans le cadre de notre recherche, nous utilisons la version que le tutoriel utilise pour créer notre cluster.

[11] http://www.oracle.com/technetwork/systems/hands-on-labs/hol-setup-hadoop-solaris-2041770.html

Le système Oracle Solaris 11.1 offre une fonctionnalité lui étant propre et qui nous sera très utile, soit le principe de « Zone ». Cela permet à un utilisateur de créer un environnement d'exécution divisé et isolé du système d'exploitation hôte. Il est important de noter qu'il est possible de se connecter à l'intérieur de l'environnement via des lignes de commande et cette action n'aura aucun impact sur le système d'exploitation qui l'héberge. Le principe fournit un environnement sécuritaire contre les attaques externes et internes d'application malveillante. De plus, chaque zone possède son propre gestionnaire de ressources, ce qui lui permet d'allouer des ressources processeur, mémoire, réseau et de stockage. Si le compte d'utilisateur est de type administrateur, il sera alors possible de gérer les zones tel que les besoins le dictent. Toutefois, il est important de savoir que ce principe existe seulement avec les nouveaux systèmes d'exploitation Oracle Solaris.

En somme, on peut vulgariser le principe de zone comme étant une machine virtuelle au sein du système d'exploitation de façon native. Toutefois, les zones ne possèdent pas d'interface graphique et sont seulement utilisables via la ligne de commande à partir du terminal du système d'exploitation hôte.

7.3.4 Exécution de tâche sur un cluster

À la suite de la création d'un cluster Hadoop, nous pouvons maintenant définir les lignes de commandes et manipulations obligatoires pour exécuter une tâche MapReduce. Il est important de se rappeler que pour exécuter une tâche, il faut que cette dernière soit sous forme de fichier de type « jar » exécutable. Par la suite, il faut effectuer le transfert du

fichier dans la zone mise en place pour Hadoop. Ensuite, on doit donner au compte *hadoop* le droit de l'utiliser et ensuite lui attribuer les bonnes permissions d'accès. De plus, il faut transférer les données que la tâche utilisera pour effectuer son traitement. Voici les lignes de commandes correspondant à ces opérations :

Transfert du Jar :
cp ~max/Downloads/LTLValidatorHadoop.jar /usr/local/hadoop

Donner les droits à hadoop :
chown -R hadoop:hadoop /usr/local/hadoop/LTLValidatorHadoop.jar

Donner les bonnes permissions:
chmod -R 755 /usr/local/hadoop/LTLValidatorHadoop.jar

Copier le fichier de données dans le bon répertoire :
hadoop dfs -copyFromLocal /usr/local/hadoophol/Traces/linesShort.xml /input-data

Par la suite, l'utilisateur peut lancer sa tâche MapReduce en utilisant une autre ligne de commande. Elle contient le nom du fichier jar à utiliser et son chemin. Toutefois, il est important de toujours spécifier les paramètres dont ce fichier a besoin. Par exemple :

hadoop jar /usr/local/hadoop/LTLValidatorHadoop.jar -i/input-data/lines505mo.xml -p1 -o/output-data/output505moP1 -t 9025596

Finalement, pour pouvoir lire les informations dans un seul fichier, il est important de regrouper les résultats en lançant la commande suivante :

hadoop dfs -getmerge /output-data/outputShortP1 /export/output/

CHAPITRE 8

TESTS DE PERFORMANCE ET COMPARAISONS

8.1 Environnement d'exécution

L'environnement d'exécution que nous avons utilisé pour effectuer les tests est une machine virtuelle ayant comme système d'exploitation Oracle Solaris 11.1. Pour créer notre cluster, nous avons utilisé le principe de zone décrit plus haut et nous avons utilisé le tutoriel du site d'Oracle. L'ordinateur sur lequel la machine virtuelle s'exécute possède 24 Go de mémoire vive et un processeur de quatre cœurs. Le cluster est constitué de cinq zones complètes et prêtes au travail. Il y a le nœud Maître, ainsi que son nœud Maître de secours identique au nœud Maître, ainsi que les trois nœuds travailleurs.

8.2 Les environnements de tests

Dans le cadre de nos recherches, nous avons comparé les deux capacités de traitements de nos algorithmes de traitements LTL. Il est important de noter que les deux utilisent la même machine de test, et qu'elles utilisent les mêmes données pour effectuer leur traitement.

8.2.1 Mr Sim

Nous avons utilisé dans nos tests les deux modes de cet outil, soit le mode séquentiel et le mode parallèle. Donc, il existe une version de l'algorithme LTLValidator utilisant le flux de travail parallèle et elle porte le nom de « ParaLTLValidation ». Nous ferons référence à cette version dans les résultats comme étant Mr Sim parallèle. Nous avons également testé l'algorithme pour fins de comparaison. La version séquentielle sera dénommée dans les résultats comme étant Mr Sim séquentiel.

8.2.2 Hadoop

Dans le cadre de l'analyse des résultats, nous ferons référence à notre algorithme LTLValidator en mode Hadoop, sous le nom de Hadoop. Il est important de noter que cette tâche est constituée de plusieurs séquences de phases et que cette dernière s'adapte à la formule à traiter.

8.3 Formules LTL

Pour nous permettre de pouvoir évaluer le temps d'exécution de l'algorithme de validation MapReduce, nous avons utilisé un ensemble de données constituées de trace d'événements générées de façon aléatoire. Chaque événement peut être constitué d'un maximum de dix paramètres aléatoires, étiquetés de p0 à p9, avec leurs valeurs associées. La longueur totale de chaque trace est comprise entre 1 et 100 000 événements et nous avons produit 500 de ces traces. Au total, cet ensemble de données s'élève à plus d'un gigaoctet de données d'événements générés aléatoirement.

Dans le cadre de nos tests, nous utilisons quatre propriétés avec une complexité qui augmente d'une formule à l'autre et nous les appliquons sur les traces préalablement générées. Voici la présentation et la description de ces propriétés :

La propriété 1 est **G** p0 ≠ 0 affirme simplement que dans chaque événement, le paramètre p0 lorsqu'il est présent, n'est jamais égal à zéro.

La propriété 2 est **G** (p0 = 0 → **X** p1 = 0), elle exprime le fait que chaque fois que p0 = 0 dans un événement, le prochain événement est tel que p1 = 0.

La propriété 3 est une généralisation de la propriété 2, soit :

$$\forall\ x \in [0\ ,\ 9] : \mathbf{G}\ (p0 = x \rightarrow \mathbf{X}\ p1 = x)$$

Cette propriété affirme que tout ce que peut prendre p0 sera pris par p1 dans le prochain événement. Les quantificateurs universels et existentiels sont utilisés simplement comme une notation abrégée. La formule de LTL réelle à valider est la conjonction logique du modèle précédent pour toutes les valeurs possibles de x entre 0 et 9, et se lit :

$$(\mathbf{G}\ (p0 = 0 \rightarrow \mathbf{X}\ p1 = 0)) \wedge (\mathbf{G}\ (p0 = 1 \rightarrow \mathbf{X}\ p1 = 1)) \ldots$$

Finalement, la propriété 4 impose que certains paramètres p_m alternent entre deux valeurs possibles. C'est-à-dire qu'elle s'évalue à vrai lorsque la valeur de p_m à l'événement courant a la même la valeur que dans deux événements à partir de celui en cours, et s'écrit:

$$\exists\ m \in [0\ ,\ 9] : \forall\ x \in [0\ ,\ 9] : \mathbf{G}\ (\ p_m = x \rightarrow \mathbf{X}\ \mathbf{X}\ \ p_m = x\ )$$

Il est important de noter, encore une fois, que les quantificateurs sont employés comme une notation abrégée.

8.4 Résultats

Pour tester nos différentes versions de l'algorithme de validation de traces, nous avons généré des fichiers aléatoires. Il est important de noter que les fichiers de traces générés sont dans le format XML traditionnel. Par la suite, dans un but de pouvoir faire une comparaison approprié, nous avons utilisé un outil spécial permettant de transformer le XML traditionnel en format Hadoop MapReduce. La taille des fichiers varie de dix événements à plus de neuf millions. Dans le cas du fichier de trace de neuf millions, la taille du fichier est presque d'un Go. Toutefois, il est important de noter qu'en raison de grande complexité de la propriété 4, cette dernière cause une erreur fatale pour les deux plus grands fichiers : ces derniers contiennent 3,5 et 9 millions d'événements.

8.4.1 Nombre de tuples

L'élément le plus important à mesurer pour nous donner une idée de la complexité du traitement à effectuer est le calcul du nombre de tuples produits. Cette valeur est calculée comme étant la somme de T_i, qui est le nombre total de tuples traités par la phase Map de chaque cycle MapReduce *i*, pour tous les cycles $i \in [1,\delta(\varphi)]$, ainsi que présenté dans le tableau 3. On peut s'apercevoir que le nombre de tuples augmente avec la complexité de la formule. Ainsi, alors que la propriété 1 génère 180 062 tuples, pour la même trace avec la propriété 4, le nombre de tuples générés est de plus de 8 millions.

Taille de la Trace	P1	P2	P3	P4
11	14	17	20	221
180,035	180,062	346,642	815,449	8,194,477
1,770,922	1,770,938	3,409,468	8,010,419	80,552,405
3,523,211	3,523,218	6,782,937	15,938,586	> 54 M
9,025,596	9,025,603	17,375,057	40,830,370	---
Ratio	1	1,9	4,5	45

Tableau 3 : Total de tuples produits par l'algorithme pour chaque propriété sur chaque fichier

Ensuite, on peut s'apercevoir que lors de l'exécution la distribution des tuples à travers les différentes itérations n'est pas uniforme. Le tableau 4 nous montre le nombre de tuples produits par une phase Reducer de chaque cycle pour chaque propriété et deux traces différentes.

Itération	P1	P2	P3	P4
1	8	8	9	11
2	3	3	4	43
3	3	3	4	42
4		3	3	59
5				34
6				24
7				8

Itération	P1	P2	P3	P4
1	132,392	132,392	662,013	856,275
2	1,638,530	1,638,530	3,674,133	26,563,830
3	16	1,638,530	3,674,133	26,563,830
4		16	47	26,564,584
5			31	1,508
6			31	1,022
7			31	750
8				336
9				28
10				28
11				54
12				80
13				80

Tableau 4 : Nombre de tuples produits lors de la phase Reducer

Par la suite, pour nous donner un autre point de vue important, nous avons calculé ce que nous appelons le « ratio séquentiel » du processus de validation. À chaque cycle MapReduce, nous gardons le plus grand nombre de tuple traités par une seule instance de Reducer. Cette valeur, notée t_i, représente le nombre minimum de tuples qui doit être traité séquentiellement dans ce cycle particulier. Si tous les Reducers pour ce cycle étaient autorisés à s'exécuter en parallèle et si nous supposons que le temps de traitement est similaire pour chaque tuple, le ratio t_i/T_i est un indicateur du temps de cycle « parallèle » basé sur celui de la version « séquentielle ». Le rapport séquentiel global indiqué dans le tableau 5, se calcule comme suit :

$$s = \frac{\sum_{i=1}^{\delta(\varphi)} t_i}{\sum_{i=1}^{\delta(\varphi)} T_i}$$

Le ratio séquentiel nous permet de déterminer les limites de l'algorithme de validation et nous permet de déterminer également le potentiel de parallélisme de ce dernier. Il est important de noter que le potentiel pour le parallélisme est délimité par la structure de la formule à valider, puisqu'il peut y avoir plus d'une instance de réducteur pour chaque sous-formule possible de la propriété à vérifier. Par conséquent, pour des formules simples, telles que les propriétés 1 et 2, qui ont très peu de sous-formules différentes à chaque cycle MapReduce, presque tout le travail doit être fait de manière séquentielle (100 % dans le cas de la propriété 1 et 92 % dans la propriété 2).

Cependant, dès que la propriété devient plus complexe, comme dans le cas de la propriété 3 et 4, la situation change du tout au tout et chaque Reducer gère une petite partie du nombre total de tuples. Dans les faits, la propriété 4 est spectaculaire à cet égard, puisque 97 % des tuples concernés peuvent être traités en parallèle. La présence de quantificateurs est en grande partie responsable de ce phénomène, car la taille de la formule de LTL réelle passée au validateur de trace grandit rapidement. Cinquante copies de la même formule sont évaluées, avec diverses combinaisons de valeurs pour m et x.

	Propriété 1	Propriété 2	Propriété 3	Propriété 4
Ratio Séquentiel	100 %	92 %	19 %	3 %

Tableau 5 : Ratio séquentiel de chaque propriété étudiée

8.4.2 Temps d'exécution

La seconde mesure que nous allons observer est le temps de fonctionnement total nécessaire à la validation de chaque propriété, sur des traces de taille croissante. Dans le tableau 6, nous présentons pour chaque propriété, la durée de fonctionnement moyenne pour chaque événement. C'est-à-dire le temps total de traitement divisé par le nombre d'événements dans la trace. Les valeurs indiquées dans le tableau sont la moyenne de ces valeurs sur toutes les traces.

À partir du ratio séquentiel *s* et du temps d'exécution moyen par événement *r* obtenu pour chaque propriété, il est également possible d'en déduire le temps de validation théorique dans le cas du traitement maximum parallèle en calculant *r x s.* Dans le tableau 6, il est indiqué la durée présumée, en utilisant l'exécution séquentielle de Mr Sim comme référence.

Le taux de croissance varie largement d'un outil à l'autre, Mr Sim parallèle étant le pire. Nous avons pu observer le fait étonnant suivant, la version multi-thread de Mr Sim et la version Hadoop ont affiché un temps de validations plus lent que la version séquentielle de Mr Sim. C'est-à-dire qu'en fonction des propriétés, ces outils sont parfois 40 fois plus lents que la version séquentielle de Mr Sim. Par la suite, au niveau du temps de traitement estimé à partir du ratio séquentiel (dernière ligne du tableau 6), on peut s'apercevoir qu'ils sont aussi parfois plusieurs ordres de grandeur plus faibles que les temps d'exécution réels observés avec la version multi-thread de Mr Sim. Il est important de noter que Mr Sim

séquentiel utilise la même tâche Map et Reduce, donc le problème ne se situe pas au niveau de l'algorithme lui-même, mais plutôt dans l'introduction du parallélisme.

Total	P1	P2	P3	P4
Mr Sim séquentiel	1.9	3.6	10.4	335
Hadoop	44	96	243	4,694
Mr Sim parallèle	73	138	433	5,734
Mr Sim prédit	1.9	3.3	2.0	10

Tableau 6 : Le temps moyen d'exécution en millisecondes pour chaque propriété et de chaque outil

CONCLUSION

Dans les chapitres précédents, nous avons pu voir comment définir le traitement de la logique voulue en utilisant la LTL. Par la suite, nous avons abordé le principe MapReduce avec ses différentes phases et la façon de les utiliser dans le cadre de nos recherches. Nous avons vu également comment écrire et traiter les équations à analyser. Sans oublier les outils complémentaires permettant de faciliter notre recherche et la façon de créer un cluster.

Finalement, nous pouvons conclure qu'il est possible d'utiliser un environnement Hadoop MapReduce pour effectuer la vérification de propriétés LTL. Il faut bien sûr faire des ajustements à la logique de traitement pour que cette dernière soit compatible avec les différentes phases MapReduce. Par la suite, il faut définir le format d'entrées des données pour permettre de bien lire les données lors de la lecture de l'InputReader, sans oublier de comprendre et d'utiliser à bien les ressources de l'environnement. Ce projet de recherche effectué avec succès, nous permet d'affirmer qu'il est viable et intéressant d'utiliser un environnement Hadoop pour effectuer le traitement de formules LTL.

Cependant, il est important de se pencher sur la définition de ce que l'on entend par cluster. Nous pouvons affirmer que le cluster que nous avons utilisé, un cluster de référence, qui contient un regroupement de 4 nœuds travaillait ensemble pour le bien de la tâche est inadéquat. Bien sûr, il nous permet d'effectuer de la recherche avec succès et nous permet d'expérimenter notre tâche. Toutefois, pour mener des tests de performance ou

d'exécution véridique, notre cluster est très surchargé pour les plus grandes traces et commence à être surchargé à partir des traces moyennes. Nous croyons que, pour faire exécuter à bien sur ces traces les propriétés 3 et 4, (nos propriétés les plus complexes), il nous aurait fallu plus de nœuds de traitements. Ceci implique qu'il aurait fallu disposer de quelques machines supplémentaires, semblables à celle utilisée, pour avoir un cluster acceptable.

En somme, en considérant nos environnements, nos propriétés et nos traces, nous pouvons conclure que notre algorithme fournit des résultats intéressants et démontre sa puissance. Il nous prouve, fait le plus important, qu'il est possible d'effectuer de l'analyse de formule LTL en parallèle avec plusieurs unités de traitement.

BIBLIOGRAPHIE

1. Halle, S. and R. Villemaire, *Runtime Enforcement of Web Service Message Contracts with Data.* IEEE Transactions on Services Computing. **5**(2): p. 192-206.
2. Groce, A., K. Havelund, and M. Smith, *From scripts to specifications: the evolution of a flight software testing effort*, in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. 2010, ACM: Cape Town, South Africa. p. 129-138.
3. Havelund, G.R.a.K., *Rewriting-based techniques for runtime verification.* Autom. Softw. Eng., 2005: p. 12(2):151–197.
4. Naldurg, P., K. Sen, and P. Thati, *A Temporal Logic Based Framework for Intrusion Detection*, in *Formal Techniques for Networked and Distributed Systems – FORTE 2004*, D. Frutos-Escrig and M. Núñez, Editors. 2004, Springer Berlin Heidelberg. p. 359-376.
5. Basin, D., F. Klaedtke, and S. Müller, *Policy Monitoring in First-Order Temporal Logic*, in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Editors. 2010, Springer Berlin Heidelberg. p. 1-18.
6. Cimatti, A., et al., *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*, in *Computer Aided Verification*, E. Brinksma and K. Larsen, Editors. 2002, Springer Berlin Heidelberg. p. 359-364.
7. Verbeek, H.M.W., et al., *XES, XESame, and ProM 6*, in *Information Systems Evolution*, P. Soffer and E. Proper, Editors. 2011, Springer Berlin Heidelberg. p. 60-75.
8. Barringer, H., D. Rydeheard, and K. Havelund, *Rule Systems for Run-time Monitoring: from Eagle to RuleR.* Journal of Logic and Computation, 2010. **20**(3): p. 675-706.
9. Hallé, S. and R. Villemaire, *XML Methods for Validation of Temporal Properties on Message Traces with Data*, in *On the Move to Meaningful Internet Systems: OTM 2008*, R. Meersman and Z. Tari, Editors. 2008, Springer Berlin Heidelberg. p. 337-353.
10. Garavel, H. and R. Mateescu, *SEQ.OPEN: A Tool for Efficient Trace-Based Verification*, in *Model Checking Software*, S. Graf and L. Mounier, Editors. 2004, Springer Berlin Heidelberg. p. 151-157.
11. Holzmann, G. and S.M. Checker, *The SPIN Model Checker: Primer and Reference Manual*. 2004, Addison Wesley Professional.
12. Pankratius, V., A. Adl-tabatabai, and W.F. Tichy, *Fundamentals of Multicore Software Development*. 2012: CRC PressINC.
13. Benjamin Barre, M.K., Maxime Soucy-Boivin, Pierre-Antoine Ollivier and Sylvain Hallé, *MapReduce for Parallel Trace Validation of LTL Properties*, in *RV 2012*. 2012: Istanbul, Turkey. p. 15 p.

14. Pnueli, A. *The temporal logic of programs*. in *Foundations of Computer Science, 1977., 18th Annual Symposium on*. 1977.
15. Dean, J. and S. Ghemawat, *MapReduce: simplified data processing on large clusters*. Commun. ACM, 2008. **51**(1): p. 107-113.
16. Kuhtz, L. and B. Finkbeiner, *LTL Path Checking Is Efficiently Parallelizable*, in *Automata, Languages and Programming*, S. Albers, et al., Editors. 2009, Springer Berlin Heidelberg. p. 235-246.
17. Pnueli, A. and A. Zaks, *On the Merits of Temporal Testers*, in *25 Years of Model Checking*, O. Grumberg and H. Veith, Editors. 2008, Springer Berlin Heidelberg. p. 172-195.

ANNEXE 1

TUTORIEL D'UTILISATION LTLVALIDATOR

Ligne de Commande LTLValidator

Dans l'annexe 1, nous présentons un exemple expliqué de la ligne de commande nécessaire pour lancer l'outil. De plus, il sera listé les paramètres possibles avec une description. Voici l'exemple avec lequel on travaillera :

-i I:\tracesHadoop\TS\TS_196mo.xml -p2 -r I:\tracesHadoop\Article\nbreEvents\LTL_Test196moP4.xml.txt -t Sax -v 3

Variable	Explication + Paramètre(s)
-i	Cette dernière permet de spécifier la source des données à étudier. Autrement dit, le fichier avec son arborescence complète. Exemple :-i I:\tracesHadoop\TS\TS_196mo.xml
-p	La variable permet de spécifier la formule à analyser, soit une formule LTL écrite par l'utilisateur ou bien d'une formule pré-écrite contenue dans le fichier «Edoc2012Presets.java». Cela permet alors à l'application de pouvoir déterminer le choix de formule de l'utilisateur. Les paramètres possibles sont : 1 – 2 – 3 – 4 ou une formule LTL Exemple : -p 4
-r	Permet à l'application de savoir où placer le fichier de résultat ainsi que son nom. Il est important de noter que si le fichier existe, alors les anciennes informations sont détruites. Sinon, si le fichier n'existe pas, alors l'outil le créera avant d'effectuer une opération d'écriture. Toutefois, si cette variable est absente, alors, l'ensemble des informations sera affiché dans la console. Exemple : I:\tracesHadoop\Article\nbreEvents\LTL_Test196moP4.xml.txt
-t	Cette variable permet de spécifier le type de parser que l'outil utilisera pour parcourir les données entrantes. Les paramètres sont soit : Dom ou Sax N.B. : Si l'utilisateur ne respecte par le format de l'écriture, l'outil utilisera le parser Sax par défaut.
-v	La variable v représente la verbosité de l'application. C'est-à-dire le niveau des informations affichées ou écrites dans le fichier. Donc, si la verbosité est à 1, alors il y aura uniquement les informations finales d'accessible pour l'utilisateur. Par la suite, le niveau 2 affichera les informations brèves sur les étapes de traitements. Sinon, avec le niveau 3 l'outil donnera accès à l'ensemble des informations pour lequel il a été programmé de donner.

ANNEXE 2

TUTORIEL D'UTILISATION XMLTRACEGENERATOR

Ligne de Commande XMLTraceGenerator

Dans l'annexe 2, un exemple de ligne de commande concrète pour lancer l'outil est présenté. Il est important de noter que les paramètres possibles sont listés et qu'une description est également fournie. Voici un exemple :

-c I:\tracesHadoop\conditions.xml -d 4 -f I:\tracesHadoop\TestTG.xml -g HadoopLTLValidator -s 538968064 -t Numbers -v 9 -w 21 -z 0

Variable	Explication + Paramètre(s)
-c	Cette variable permet de spécifier l'arborescence ainsi que le fichier contenant les conditions. Ces dernières permettent de déterminer toutes les valeurs possibles pour toutes les variables possibles.
-d	Cette variable permet de contenir la profondeur maximum qu'un événement peut avoir. C'est-à-dire le nombre de variables contenu par tous les événements varieront entre 0 et la valeur placée en paramètre. Il est important de noter que c'est l'utilisateur qui choisit la valeur. Toutefois, la profondeur doit être différente de zéro, puisqu'il est inutile de générer un ensemble complet d'événements vides.
-f	La variable f permet de contenir l'arborescence ainsi que le fichier permettant de contenir les tags XML créés selon les conditions déterminées par l'utilisateur.
-g	Cette variable permet de contenir le type de générateur à utiliser. Dans le cas du générateur HadoopTraceGenerator, la valeur de cette variable doit être «HadoopLTLValidator». Sinon pour le générateur XMLTraceGenerator la valeur est « LTLValidator ».
-s	La variable -s permet de contenir la taille totale physique que le fichier doit tendre à devenir en octets. C'est-à-dire que le fichier générera des événements et les enregistrera dans le fichier tant que la taille totale n'est pas égale ou supérieure.
-t	Cette variable permet de déterminer le type de valeur que contiendront les variables des événements. C'est-à-dire que si le paramètre possède la valeur «Numbers», alors les valeurs seront toutes des nombres. Par la suite, le paramètre «Letters» permettra de générer des variables avec des lettres comme valeur. Ensuite, le paramètre «Both» permet de générer des événements avec des variables numériques et alphabétiques. Toutefois, le type de la valeur est choisi de façon aléatoire. Pour ce faire, l'application choisit une valeur aléatoire entre 0 et 100. Ensuite, si la valeur est inférieure à 50, alors le type sera numérique, sinon il sera alphabétique. Le type par défaut est numérique si le

	paramètre est inconnu.
-v	Cette variable permet de déterminer le nom complet de la variable. Il est important de noter que les variables sont du format «p numéro». La variable permet de déterminer l'étendue possible des numéros contenue dans les noms de variables. Dans les faits, la variable varie de 0 au paramètre choisi.
-w	La variable v permet de déterminer la profondeur de la valeur des variables. Si le paramètre de la variable t est numérique, la valeur varie de 0 à la valeur du paramètre et le choix est fait de façon aléatoire. Sinon, pour une valeur alphabétique, le chiffre choisi est diminué de 24, tant que celui-ci n'est pas inférieur à 25. Par la suite, le chiffre est remplacé par la lettre correspondante. Sinon, si le type est « Both », la valeur inscrite se basera sur le choix aléatoire fait, présenté lors de la description du paramètre t.
-z	La variable v représente la verbosité de l'application. C'est-à-dire le niveau des informations affichées ou écrites dans le fichier. Donc, si la verbosité est à 1, alors il y aura uniquement les informations finales d'accessible pour l'utilisateur. Par la suite, le niveau 2 affichera les informations brèves sur les étapes de traitements. Sinon, avec le niveau 3 l'outil donnera accès à l'ensemble des informations pour lequel il a été programmé de donner.

ANNEXE 3

TUTORIEL D'UTILISATION TAGCOUNTER

Ligne de Commande TagCounter

Dans l'annexe 3, nous présentons une ligne de commande, donc un exemple concret sur la façon de paramétrer l'outil. Par la suite, il y a des explications et la liste de paramètres de chaque variable qui doit être utilisée. Voici un exemple :

-t p0 -i \traces\196mo.xml -r C:/jobs/results/Results_exec1.bat.txt

Variable	Explication + Paramètre(s)
-t	Cette variable contient le tag à compter à travers l'ensemble du fichier. Les paramètres possibles de cette variable sont infinis, puisqu'il peut prendre n'importe quelle valeur.
-i	Cette variable permet de spécifier la source des données à étudier. Autrement dit, le fichier avec son arborescence complète. Exemple : -i \traces\196mo.xml
-r	Cela permet à l'application de savoir où placer le fichier de résultat ainsi que son nom. Il est important de noter que si le fichier existe, alors les anciennes informations sont détruites. Sinon, si le fichier n'existe pas, alors l'outil le créera avant d'effectuer une opération d'écriture. Exemple : C:/jobs/results/Results_exec1.bat.txt

ANNEXE 4

TUTORIEL D'UTILISATION JOBLAUNCHER

Ligne de Commande JobLauncher

Dans l'annexe 4, un exemple de ligne de commande est présenté pour permettre d'utiliser l'outil avec succès. Par la suite, il y a dans le tableau des explications et paramètre(s) de chaque variable qui doit être utilisée. Voici un exemple concret d'une ligne de commande :

-b C:\tag\bat -o C:\tag\results -u Émetteur -p MotDePasse -r shalle@acm.org -e true

Variable	Explication + Paramètre(s)
-b	La variable b permet de déterminer l'arborescence ou se trouve le dossier contenant l'ensemble des fichiers bat à exécuter. C'est-à-dire que chaque fichier contient la ligne de commande nécessaire pour faire exécuter le jar exécutable voulu.
-o	Cette variable permet de déterminer le dossier qui contiendra, s'il y a lieu, les fichiers de sorties suite à l'exécution des fichiers bat. Le nom du fichier de sortie est crée sous le nom «Results_» suivi du nom complet du fichier bat et de l'extension «.txt».
-u	La variable u permet de spécifier l'adresse courriel Gmail qui sera utilisée pour envoyer les avertissements de fin d'exécution. Pour ce faire, l'outil se connecte via smtp au serveur de Google et lui transfert un objet courriel avec toutes les informations nécessaires. À la fin de l'opération de l'envoi de l'objet, la connexion est fermée.
-p	Cette variable permet de contenir le mot de passe de l'adresse courriel émettrice des avertissements par courriel. Il est important de noter que le mot de passe est écrit en clair. C'est-à-dire que ce dernier n'est pas encrypté.
-r	La variable r permet de spécifier la ou les adresses courriels du ou des destinataires de l'ensemble des traitements en lot.
-e	Cette variable permet de déterminer si la fonctionnalité des avertissements par courriel est active ou inactive. Soit égale à «true» ou «false».

ANNEXE 5

TUTORIEL DE CRÉATION D'UN ENVIRONNEMENT DE DEBUGAGE DE JOB HADOOP

Comment créer un environnement de débogage de job Hadoop avec un OS Ubuntu
Tutoriel inspiré par celui de :

Auteur : *Praveen Sripati*

Site web : http://www.thecloudavenue.com/2012/10/debugging-hadoop-mapreduce-program-in.html

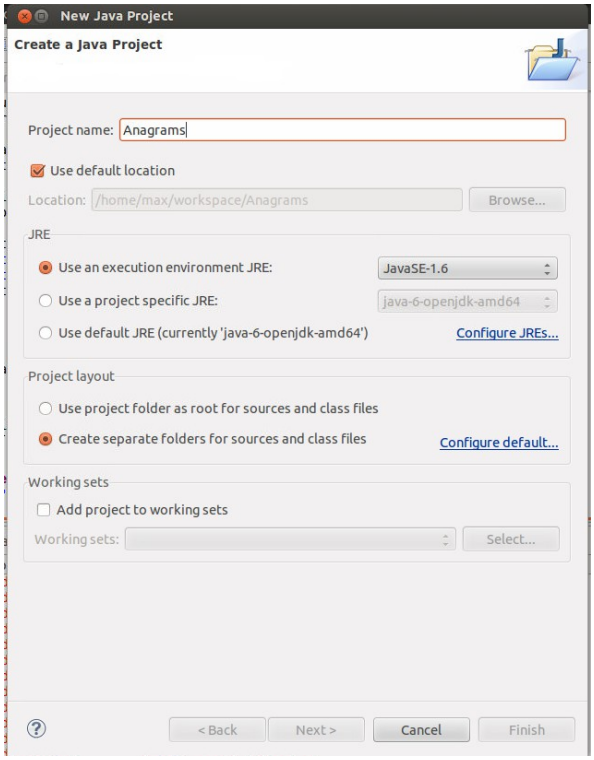
Sommaire: Tutoriel permettant d'établir un environnement permettant de déboguer pour l'ensemble de la tâche. Il est alors possible d'accéder à l'exécution de Mapper, à celui du Reducer, comme toute application java. Pour ce faire, il faut tout simplement utiliser les points d'arrêts à l'intérieur de ceux-ci. Il est important de noter que l'environnement créé contient uniquement un nœud de traitement. Alors, le parallélisme est quasiment inexistant, mais cela permet de voir comment l'ensemble des nœuds réagiront au code.

1. Il faut commencer par avoir un système d'exploitation Ubuntu de prêt pour pouvoir commencer à créer un environnement de débogage. Donc, si ce n'est pas le cas, vous devez télécharger l'image sur le site officiel et l'installer. La version qui est conseillée pour ce tutoriel est Ubuntu 12.04 LTS. Si jamais, vous ne voulez pas installer le système d'exploitation de façon native sur votre ordinateur, vous pouvez vous créer une machine virtuelle pour ce tutoriel.

   Si jamais vous êtes un utilisateur Windows, il est important de savoir que ce tutoriel ne fonctionnera pas de façon native, parce que nous utilisons des utilitaires n'existant pas dans aucune distribution de Microsoft. Pour ceux qui ne jurent que par Microsoft, vous pouvez installer l'application Cygwin. Toutefois, le fonctionnement de ce dernier n'est pas garanti.

2. Ensuite, vous devez installer l'environnement de développement Éclipse. Il est important de noter qu'il est peut-être possible d'utiliser un autre IDE de développement. Toutefois, vous devrez vous-même trouver la façon d'adapter ce tutoriel à celui-ci. Il est important de savoir que la version d'Éclipse utilisée pour ce tutoriel est 3.7.2.

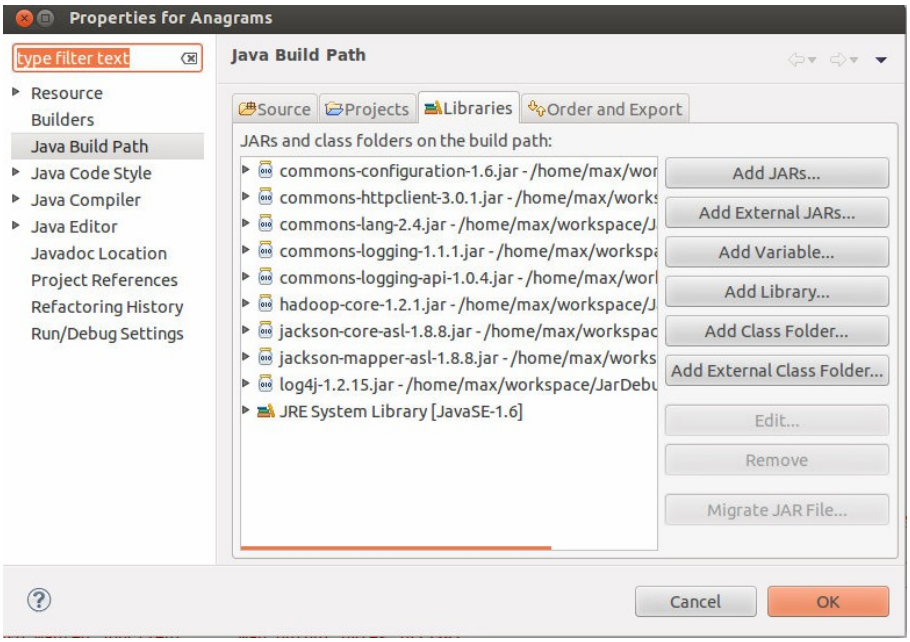
3. Il faut ensuite créer un nouveau projet Java en faisant « File/New/Java Project ». Le nom de notre projet est Anagrams.



4. Par la suite, vous devez télécharger le code source de Hadoop. Il est important de noter que la version utilisée et courante est 1.2.1. Le site web pour télécharger la dernière et ou version d'archive est celui d'apache[12].

[12] http://www.apache.org/dyn/closer.cgi/hadoop/common/

5. Ensuite, vous devez intégrer les dépendances suivantes à votre projet : commons-configuration-1.6.jar, commons-httpclient-3.0.1.jar, commons-lang-2.4.jar, commons-logging-1.1.1.jar, commons-logging-api-1.0.4.jar, hadoop-core-1.2.1.jar, jackson-core-asl-1.8.8.jar, jackson-mapper-asl-1.8.8.jar et log4j-1.2.15.jar. Pour ce faire, vous devez faire un clic de droit sur votre projet et cliquer sur «Build Path/Configure Build Path …». Ensuite, vous devez cliquer sur «Add External JARs…» et vous aller chercher les fichiers jar qui contient les dépendances voulues et vous cliqué sur «ok».



6. Par la suite vous devez vous créer une classe Anagrams et copier le code suivant à l'intérieur de celle-ci.

Voici la source du code : Source : www.hadoopmaterial.com/2013/10/anagrams.html

```
import org.apache.hadoop.mapred.MapReduceBase;

import java.io.IOException;
import java.util.Arrays;
import java.util.Iterator;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

import org.apache.hadoop.mapred.Reducer;
import java.util.StringTokenizer;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;

public class Anagrams {
    /**
    * The Anagram mapper class gets a word as a line from the HDFS input and
    * sorts the letters in the word and writes its back to the output collector
    * as Key : sorted word (letters in the word sorted) Value: the word itself
    * as the value. When the reducer runs then we can group anagrams together
    * based on the sorted key.
```

```
 */

public static class AnagramMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, Text>
{
        private Text sortedText = new Text();
        private Text orginalText = new Text();
        @Override
        public void map(LongWritable key, Text value, OutputCollector<Text,
        Text> outputCollector, Reporter reporter) throws IOException
        {
                String word = value.toString();
                char[] wordChars = word.toCharArray();
                Arrays.sort(wordChars);
                String sortedWord = new String(wordChars);
                sortedText.set(sortedWord);
                orginalText.set(word);
                outputCollector.collect(sortedText, orginalText);
        }
}

/**
* The Anagram reducer class groups the values of the sorted keys that came
* in and checks to see if the values iterator contains more than one word.
* if the values contain more than one word we have spotted a anagram.
*/

public static class AnagramReducer extends MapReduceBase implements
Reducer<Text, Text, Text, Text>
```

```
{
        private Text outputKey = new Text();
        private Text outputValue = new Text();

        @Override
        public void reduce(Text anagramKey, Iterator<Text> anagramValues,
        OutputCollector<Text, Text> results, Reporter reporter) throws IOException
        {
         String output = "";
        while (anagramValues.hasNext())
        {
                Text anagam = anagramValues.next();
                 output = output + anagam.toString() + "~";
        }
        StringTokenizer outputTokenizer = new StringTokenizer(output, "~");
        if (outputTokenizer.countTokens() >= 2)
        {
                output = output.replace("~", ",");
                outputKey.set(anagramKey.toString());
                outputValue.set(output);
                results.collect(outputKey, outputValue);
        }
  }
}

public static void main(String[] args) throws Exception
{
        if (args.length != 2) {
                System.err.println("Usage: anagrams <in> <out>");
```

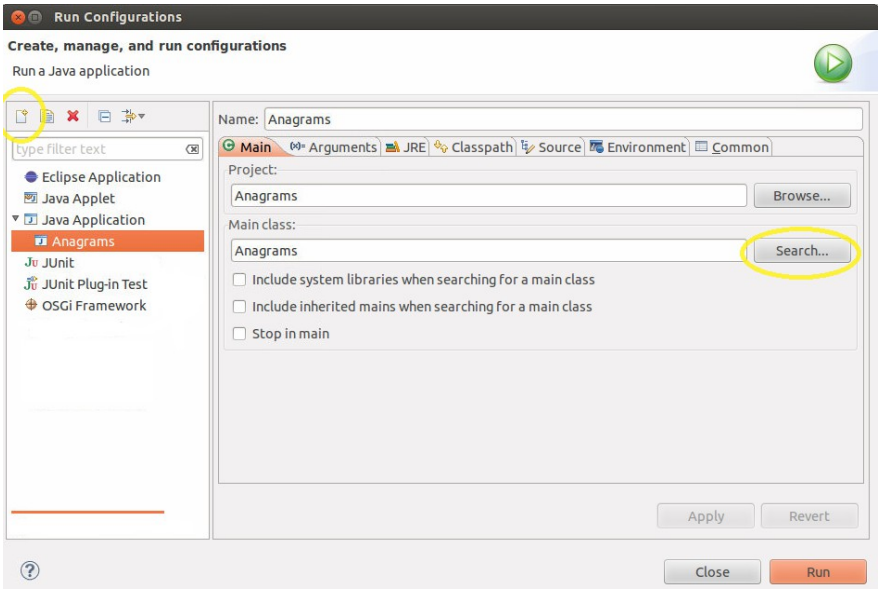
```
            System.exit(2);
        }

        JobConf conf = new JobConf(Anagrams.class);
        conf.setJobName("anagramcount");

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);

        conf.setMapperClass(AnagramMapper.class);
        conf.setCombinerClass(AnagramReducer.class);
        conf.setReducerClass(AnagramReducer.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
    }
}
```

7. Il faut ajouter un dossier au projet pour pouvoir fournir à la tâche les données à traiter. Dans ce cas-ci, nous allons utiliser le nom normalisé dans l'univers Hadoop, soit input. Pour ajouter un dossier, il suffit de faire un clic de droit « New/Folder », de taper son nom et cliquer « Finish ». Vous devez vous trouver un fichier de données ou sinon aller le chercher dans le dossier :

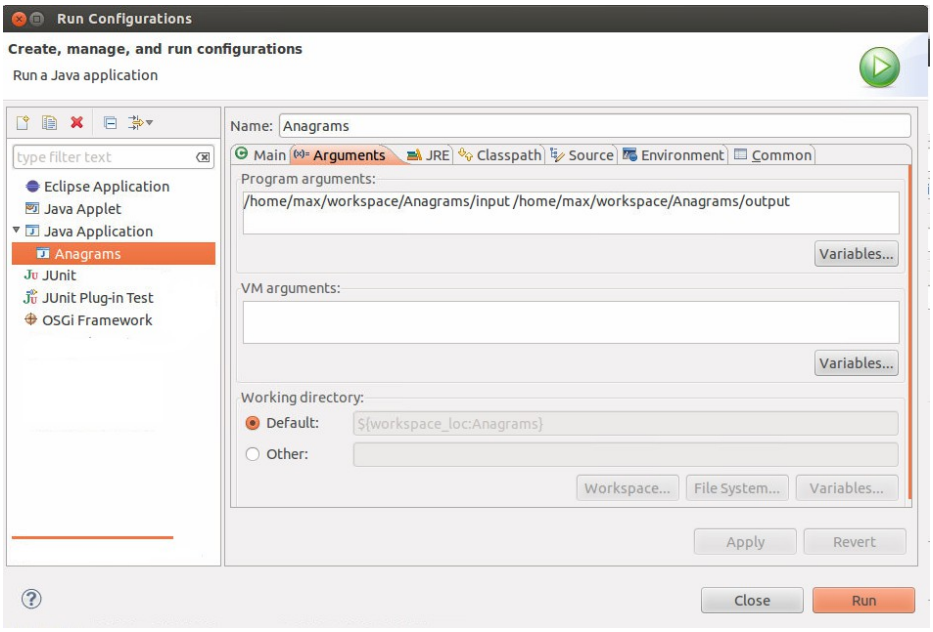
https://github.com/MaximeSoucy-Boivin/AnagramsDebugEnvironment/tree/master/Anagrams/input

8. Ensuite, vous devez ajouter les arguments aux projets. Ces derniers sont tout simplement le dossier des données entrantes et le dossier des données sortantes. Il est important de savoir que pour le dernier dossier, il n'est pas à créer avant toutes exécutions. Dans les faits, entre chaque exécution, vous devez le supprimer pour pouvoir refaire exécuter la tâche.

   Pour ajouter les arguments, vous devez cliquer sur « Run/Run Configurations… ». Ensuite, il faut créer une nouvelle configuration en cliquant en haut à gauche. Il faut également choisir la « Main Class », dans ce cas-ci Anagrams et donner les arguments.

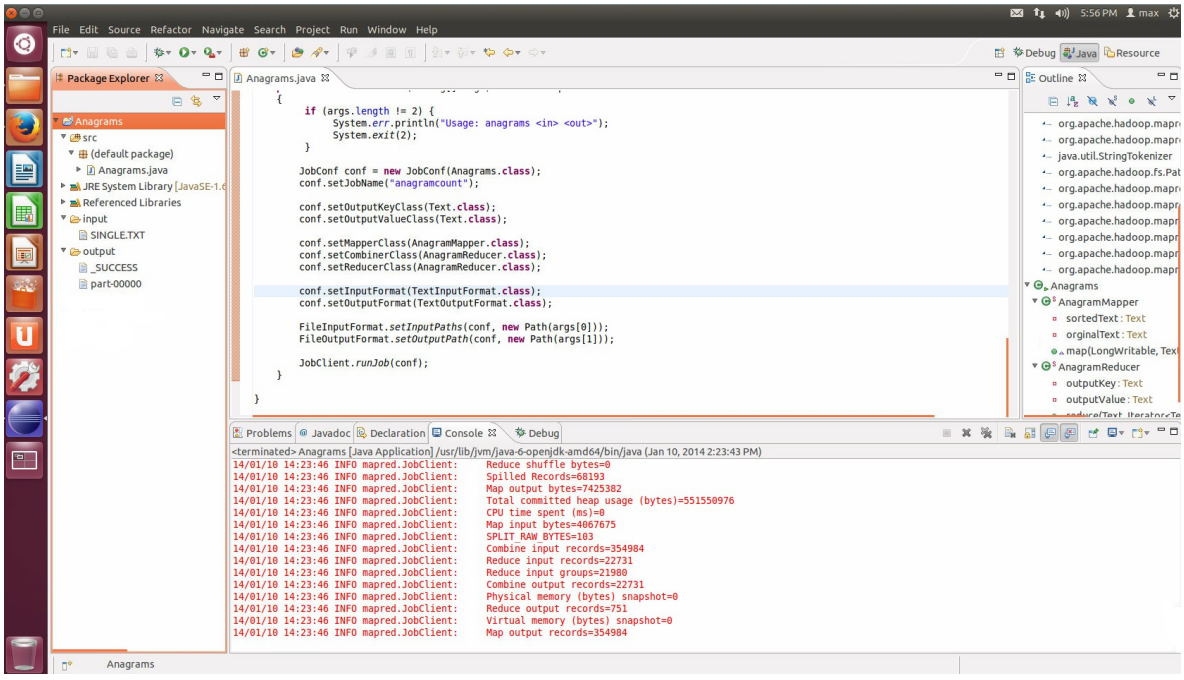


   Le chemin est celui du projet au sein du « workspace » de l'utilisateur répliqué une fois avec input et l'autre fois avec output.



9. Voilà ! Vous êtes prêts à exécuter votre première tâche Hadoop dans un environnement de débogage.

   Voici un exemple d'exécution avec succès !



10. Point d'arrêt

    Vous pouvez utiliser les points d'arrêts dans l'ensemble des différentes phases de votre tâche. La logique de fonctionnement est identique à celle d'un programme Java traditionnel. Donc, dès que le nœud est rendu à cette ligne d'exécution le programme arrête et attend l'interaction de l'utilisateur.

    Vous pouvez trouver le projet complet sur le site :
    https://github.com/MaximeSoucy-Boivin/AnagramsDebugEnvironment